

# Client-Side Web Security

## Mitigating Threats against Web Sessions

**Philippe De Ryck**

Supervisor:

Prof. dr. ir. W. Joosen

Dr. ir. L. Desmet, co-supervisor

Dissertation presented in partial  
fulfillment of the requirements for the  
degree of Doctor in Engineering

December 2014



# **Client-Side Web Security**

Mitigating Threats against Web Sessions

**Philippe DE RYCK**

Examination committee:

Prof. dr. A. Bultheel, chair

Prof. dr. ir. W. Joosen, supervisor

Dr. ir. L. Desmet, co-supervisor

Prof. dr. ir. F. Piessens

Prof. dr. ir. B. Preneel

Prof. dr. ir. Y. Berbers

Dr. M. Johns

(SAP Research, Karlsruhe, Germany)

Dissertation presented in partial  
fulfillment of the requirements for  
the degree of Doctor  
in Engineering

© 2014 KU Leuven – Faculty of Engineering Science  
Uitgegeven in eigen beheer, Philippe De Ryck, Celestijnenlaan 200A box 2402, B-3001 Heverlee (Belgium)

Alle rechten voorbehouden. Niets uit deze uitgave mag worden vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotokopie, microfilm, elektronisch of op welke wijze ook zonder voorafgaande schriftelijke toestemming van de uitgever.

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm, electronic or any other means without written permission from the publisher.

ISBN 978-94-6018-945-6  
D/2014/7515/166

# Preface

The Web has become an intrinsic part of our modern society, but unfortunately, so have security incidents. In recent years, these incidents are even being covered in mainstream media, strongly highlighting the need for effective security measures. This dissertation focuses on client-side mitigation techniques, and investigates the question whether client-side mitigation techniques can be used to effectively secure a session between the browser and a Web application.

I would like to use this preface to acknowledge the invaluable guidance and support of many people, without which none of what follows in this dissertation would have been possible.

First and foremost, I would like to express my gratitude to Lieven, my co-supervisor and daily coach. Lieven's insights and guidance have been essential to this dissertation, and to my development as a researcher in general. I very much appreciate Lieven's low-barrier availability, and his willingness to provide feedback. Lieven, thank you for shielding me from various managerial tasks and procedures, for acting upon concerns I vented informally over lunch, and for providing your valuable insights in scientific, educational and social issues.

I would also like to thank Wouter, my supervisor, for giving me the opportunity to join DistriNet, for providing a frame of reference for my research, for treating us like professionals, and for ultimately watching over me. I am also grateful to Frank, for making time for me whenever needed, and for giving me the chance to participate in various events and projects. Finally, I would also like to thank the members of my examination committee for their reflections on my dissertation, and for the interesting discussion at the defense.

I would also like to thank my collaborators and co-authors on various projects and papers for the enriching experiences. A special mention goes out to Steven, whose technical skills far exceed mine, but was always willing to help and share knowledge. I would also like to thank my colleagues at DistriNet for the enjoyable work environment, and the interesting discussions on a wide variety

of topics. I hope we will have many more of these in the future. Finally, I would also like to mention the wonderful support of the administrative staff. Katrien, Ghita, Annick, Marleen, Karen, Esther, you make our lives at the department significantly easier, thank you for that. Similarly, thank you to the systems group for keeping the infrastructure up and running, and supporting us with our sometimes unusual needs.

I am also very grateful to my parents, for catalyzing my curiosity, for enabling my education in computer science, and for the never-ending support. I would also like to thank my sister Catherine, for always supporting me, and for providing a perspective on topics in which you are infinitely smarter than me. I am also grateful to my family-in-law, for the warm environment, nutritious support, the numerous hours spent conquering mystical worlds, and the activation of skills I did not know I had.

Last, but certainly not least, I would like to thank my wife Hilde, for supporting me when I need it, for understanding when plans had to be changed once again, and for endorsing my sometimes wild ideas. Thank you for making me a better person, I hope I can do the same for you.

This research would not have been possible without the financial support of the Agency for Innovation by Science and Technology in Flanders (IWT), and iMinds. It is partially funded by the Research Fund KU Leuven, and by the EU FP7 projects STREWS, WebSand, and NESSoS.

Philippe De Ryck  
December 2014

# Abstract

As the Web has claimed a prominent place in our society and in our daily lives, Web security has become more important than ever, illustrated by the mainstream media coverage of serious Web security incidents. Over the last years, the center of gravity of the Web has shifted towards the client, where the browser has become a full-fledged execution platform for highly dynamic, complex Web applications. Unfortunately, with the rising importance of the client-side execution context, attackers also shifted their focus towards browser-based attacks, and compromises of client devices. Naturally, when the attackers' focus shifts towards the client, the countermeasures and security policies evolve as well, as illustrated by the numerous autonomous client-side security solutions, and the recently introduced server-driven security policies, that are enforced within the browser.

In this dissertation, we elaborate on the evolution from server-side Web applications to the contemporary client-side applications, that offer a different user experience. We explore the underlying concepts of such applications, and illustrate several important attacks that can be executed from the client side. Ultimately, the focus of this dissertation lies with the security of Web sessions and session management mechanisms, an essential feature of every modern Web application. Concretely, we present three autonomous client-side countermeasures that improve the security of currently deployed session management mechanisms. Each of these countermeasures is implemented as a browser add-on, and is thoroughly evaluated. A fourth technical contribution consists of an alternative session management mechanism, that fundamentally eliminates common threats against Web sessions. A thorough evaluation of our prototype implementation shows the benefits of such an approach, as well as the compatibility with the current Web infrastructure. Finally, we report on our experience with developing client-side countermeasures, both during the inception phase, often backed by theoretical approaches, including formal modeling and rigorous security analyses, and during the development phase, resulting in practically deployable solutions, for example as a browser add-on.



# Beknopte samenvatting

Aangezien het Web een prominente plaats in onze maatschappij en in onze dagelijkse levens opgeëist heeft, is de beveiliging ervan nog belangrijker dan tevoren, adequaat geïllustreerd door de vele media-aandacht voor ernstige veiligheidsincidenten op het Internet. De laatste jaren is het zwaartepunt van het Web verschoven naar de client-zijde, waar de browser een volwaardig applicatieplatform voor dynamische en complexe Webapplicaties geworden is. Helaas betekent de stijgende belangrijkheid van de uitvoeringscontext aan de client-zijde ook dat de focus van de aanvallers zich verlegt naar aanvallen vanuit de browser, vaak met nefaste gevolgen voor de volledige omgeving aan de client-zijde. Een natuurlijk gevolg is dan ook dat de tegenmaatregelen en de beleidsregels mee evolueren, uitstekend geïllustreerd door enerzijds de autonome beveiligingsoplossingen aan de client-zijde, en anderzijds de recent geïntroduceerde server-gestuurde beleidsregels, afgedwongen door de browser.

In dit proefschrift wordt de evolutie van server-gestuurde Webapplicaties naar hedendaagse Webapplicaties met een vernieuwde gebruikerservaring uitvoerig besproken. We bespreken de onderliggende concepten van zulke moderne applicaties, en illustreren de vele belangrijke aanvallen dat vanaf de client-zijde uitgevoerd kunnen worden. Na deze inleiding focust dit proefschrift zich op de beveiling van Websessies en mechanismes voor sessiebeheer, een essentieel onderdeel van elke moderne Webapplicatie. Concreet presenteren we drie autonome beschermingsmiddelen aan de client-zijde, die de veiligheid van huidige mechanismes voor sessiebeheer verbeteren. Elk van deze beschermingsmiddelen is geïmplementeerd als een browser-extensie, en is grondig geëvalueerd. Een vierde technische bijdrage bestaat uit een alternatief mechanisme voor sessiebeheer, waarin de huidige gevaren voor Websessies geëlimineerd worden. Een grondige evaluatie van het prototype toont zowel de voordelen van deze aanpak, als de compatibiliteit met de huidige infrastructuur van het Web. Tot slot bespreken we onze ervaringen met het ontwikkelen van tegenmaatregelen aan de client-zijde, met een specifieke focus op browserextensies.



# Abbreviations

ABE	Application Boundaries Enforcer
ACM	Association for Computing Machinery
AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
ARP	Address Resolution Protocol
CA	Certificate Authority
CORS	Cross-Origin Resource Sharing
CSP	Content Security Policy
CSRF	Cross-Site Request Forgery
CSS	Cascading Style Sheets
DANE	DNS-based Authentication of Named Entities
DNS	Domain Name System
DNSSEC	Domain Name System Security Extensions
DOM	Document Object Model
DVD	Digital Video Disc
EU	European Union
FTP	File Transfer Protocol
GIF	Graphics Interchange Format
HMAC	Hash-based Message Authentication Code
HSTS	HTTP Strict Transport Security
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure

ID	Identifier
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IFIP	International Federation for Information Processing
IP	Internet Protocol
IT	Information Technology
JS	JavaScript
JSON	JavaScript Object Notation
KU Leuven	Katholieke Universiteit Leuven
OS	Operating System
OWASP	Open Web Application Security Project
PDF	Portable Document Format
PFS	Perfect Forward Secrecy
PHP	HyperText Preprocessor
PKI	Public Key Infrastructure
RFC	Request for Comments
SID	Session Identifier
SLA	Service Level Agreement
SOP	Same-Origin Policy
SQL	Structured Query Language
SSL	Secure Sockets Layer
STREWS	Strategic Research Roadmap for European Web Security
TLS	Transport Layer Security
UI	User Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VM	Virtual Machine
WPA	Wi-Fi Protected Access
XML	Extensible Markup Language

XSS	Cross-Site Scripting
-----	----------------------



# Contents

<b>Abstract</b>	<b>iii</b>
<b>Contents</b>	<b>xi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 The Web at a Glance . . . . .	2
1.2 The Relevance of Client-Side Web Security . . . . .	5
1.3 Overview of this Dissertation . . . . .	8
<b>2 Background on Client-Side Web Security</b>	<b>11</b>
2.1 Building Blocks of Web Applications . . . . .	12
2.1.1 Client-Side Execution Context . . . . .	13
2.1.2 Session Management . . . . .	14
2.1.3 Cross-Origin Interactions . . . . .	16
2.1.4 Requests and Responses on the Network . . . . .	17
2.2 Threat Models . . . . .	18
2.3 Common Client-Side Attacks . . . . .	20
2.3.1 Eavesdropping Attacks . . . . .	21
2.3.2 Man-in-the-Middle Attacks . . . . .	24
2.3.3 Cross-Site Request Forgery . . . . .	28

2.3.4	UI Redressing . . . . .	34
2.3.5	Session Hijacking . . . . .	38
2.3.6	Session Fixation . . . . .	42
2.3.7	Cross-Site Scripting . . . . .	45
2.3.8	Social Engineering Attacks . . . . .	49
2.4	Contributions Revisited . . . . .	52
2.4.1	Scope . . . . .	53
2.4.2	Solutions . . . . .	55
<b>3</b>	<b>Protecting Users against Cross-Site Request Forgery</b>	<b>57</b>
3.1	Introduction . . . . .	59
3.2	Cross-Origin HTTP Requests . . . . .	61
3.2.1	Attack Scenarios . . . . .	62
3.2.2	Non-Malicious Cross-Origin Scenarios . . . . .	63
3.3	Automatic and Precise Request Stripping . . . . .	64
3.4	Formal Modeling and Checking . . . . .	68
3.4.1	Modeling our Countermeasure . . . . .	68
3.4.2	Using Model Checking for Security and Functionality . . . . .	70
3.5	Implementation . . . . .	71
3.6	Evaluating the Trusted-Delegation Assumption . . . . .	72
3.7	Related Work . . . . .	74
3.8	Conclusion . . . . .	77
<b>4</b>	<b>Preventing Session Fixation Attacks in the Browser</b>	<b>79</b>
4.1	Introduction . . . . .	81
4.2	Session Management . . . . .	83
4.2.1	Cookies as Session Identifiers . . . . .	83
4.2.2	Parameters as Session Identifiers . . . . .	84

4.2.3	Attacks on Session Management . . . . .	84
4.3	Session Fixation . . . . .	84
4.3.1	Parameter-Based Session Fixation . . . . .	85
4.3.2	Cookie-Based Session Fixation . . . . .	86
4.3.3	Current Countermeasures . . . . .	87
4.3.4	Example Attack Scenario . . . . .	88
4.4	Client-Side Protection against Session Fixation . . . . .	89
4.4.1	General Approach . . . . .	89
4.4.2	Parameter-Based Exchanges . . . . .	91
4.4.3	Session Identifier Identification . . . . .	91
4.4.4	Prototype Implementation . . . . .	92
4.5	Evaluation . . . . .	92
4.5.1	Session Identifier Identification . . . . .	93
4.5.2	Application Compatibility . . . . .	94
4.6	Discussion . . . . .	94
4.7	Conclusion . . . . .	96
<b>5</b>	<b>Upgrading the HTTP Session Management Mechanism</b>	<b>97</b>
5.1	Introduction . . . . .	99
5.2	Background . . . . .	101
5.2.1	Session Management on the Web . . . . .	102
5.2.2	Threat Model for Session Management . . . . .	103
5.2.3	Objectives for New Session Management Mechanisms .	103
5.3	SecSess . . . . .	105
5.3.1	General Idea . . . . .	106
5.3.2	Detailed Explanation . . . . .	106
5.3.3	Handling Modified Request Flows . . . . .	108

5.4	Implementation and Evaluation . . . . .	109
5.4.1	Security . . . . .	110
5.4.2	Overhead . . . . .	110
5.4.3	Compatibility with Web Caches . . . . .	112
5.5	Discussion . . . . .	113
5.5.1	Deploying SecSess . . . . .	113
5.5.2	Related Attacks . . . . .	114
5.6	Related Work . . . . .	115
5.7	Conclusion . . . . .	117
<b>6</b>	<b>Client-Side Detection of Tabnabbing Attacks</b>	<b>119</b>
6.1	Introduction . . . . .	121
6.2	Background . . . . .	123
6.2.1	Anatomy of a Tabnabbing Attack . . . . .	123
6.2.2	Overly Specific Detection . . . . .	124
6.3	TabShots Prototype . . . . .	126
6.3.1	Core Idea . . . . .	126
6.3.2	Implementation Details . . . . .	126
6.3.3	Alternative Design Decisions . . . . .	129
6.4	Evaluation . . . . .	129
6.4.1	Security . . . . .	130
6.4.2	Performance . . . . .	130
6.4.3	Compatibility . . . . .	132
6.5	Blacklisting Tabnabbing . . . . .	135
6.6	Related work . . . . .	137
6.7	Conclusion . . . . .	138
<b>7</b>	<b>Reflections on Client-side Web Security</b>	<b>141</b>

7.1	Experience Report on Client-Side Mitigations . . . . .	142
7.1.1	Positioning a Client-Side Mitigation Technique . . . . .	142
7.1.2	The Capabilities of Browser Add-ons . . . . .	145
7.1.3	Evaluating Browser Add-on Prototypes . . . . .	147
7.2	Research Challenges and Trends . . . . .	148
7.2.1	Theoretical Underpinnings of Web Security . . . . .	149
7.2.2	Upcoming State-of-Practice Security Policies . . . . .	150
7.2.3	The Rise of the Mobile Web . . . . .	152
<b>8</b>	<b>Conclusion</b>	<b>155</b>
<b>Bibliography</b>		<b>159</b>
<b>List of Publications</b>		<b>183</b>



# Chapter 1

## Introduction

Google [174], LinkedIn [193], Adobe [132], Yahoo [100], eBay [202], Nintendo [147], LastPass [146], Vodafone [150], Target [203], Reuters [138]. There may not seem to be an apparent commonality between these companies, but they have all been victims of Web-based attacks, resulting in the compromise of customer accounts, the large-scale theft of customer information or embarrassing defacements of their Web sites. The list includes ten prominent companies, that are well aware of the dangers of the Web, and they are only the tip of the iceberg. A report about Web security in 2013 lists 253 data breaches [234], good for exposing a total of 552 million identities, and reports an astonishing 568,700 Web attacks blocked *per day*. Statistics show that cybercrime makes 378 million victims per year, or 12 victims per second. Even though financial numbers on cybercrime-induced losses are very unreliable [101], Symantec estimates the direct global losses caused by cybercrime at \$113 billion in a single year, enough to host the London Olympics about 10 times over [233].

The averse effects of these Web attacks are often underestimated, both for companies and for individuals. Companies that have become victims of a data breach or defacement not only suffer from business disruptions, but also face investigations and potential lawsuits. Additionally, the ensuing reputation damage can cause long-term harmful effects, with customers leaving and shareholders losing confidence. Even worse, a continuous stream of security breaches can cause a loss of confidence in online services among the general population, severely hurting the online retail economy, e-government and e-health services.

A 2013 survey [242] reports that 70% of surveyed Internet users are concerned that their personal information is not kept secure by Web sites, resulting in

adapted behavior, as 34% of the users is less likely to give personal information on Web sites. And indeed, security breaches cause significant collateral damage to individual users. For example, a stolen database of personal information often contains users' email addresses, and maybe even recoverable passwords. If the same credentials are used for the email account, the user can lose control over this account, as well as over all accounts that are associated with that email address. Even worse, the stolen information can be used to commit identity theft, resulting in fraudulent costs being attributed to the victim, instead of the perpetrator.

In other cases, the Web attack is only used as a stepping stone towards the compromise of a larger target. For example, Belgacom, a Belgian telco also running infrastructure in Africa, was targeted by the British intelligence service GCHQ [133] through a Web attack. The attackers faked a social network application to serve malware to a Belgacom engineer, allowing the attackers to further infiltrate the Belgacom infrastructure. Another example is the 2010 compromise of *apache.org*, where a number of Web vulnerabilities eventually led to the compromise of the machine holding the code repositories [106].

With cybercrime as a billion-dollar business, the Web is in a dire situation. Web security is more important than ever, today and in the future. Before we start discussing attackers, problems, and their countermeasures, we take a closer look at how the Web came to be the way it is today, and why client-side Web security, the main focus of this dissertation, has become so popular.

## 1.1 The Web at a Glance

The World Wide Web started out as a distributed hypertext system, where documents hosted on networked computers contain references to other documents, hosted on different networked computers. These documents can be retrieved using a browser, dedicated client software for viewing hypertext documents, and following hyperlinks embedded within the text of these documents.

In order to make such a distributed hypertext system work, three fundamental agreements (standards) are necessary:

1. **Resource Identifiers** URIs [37] (originally called URLs) provide universally dereferenceable identifiers for resources on the Web.
2. **Transfer Protocol** HTTP [99] (Hypertext Transfer Protocol) is a universally supported transfer protocol that, in its bare essence, provides

a simple mechanism to retrieve a resource across the network, and to submit form data from a browser to a Web server. HTTP is a request/response-based client-server protocol: The browser will send a request to a server that (a) identifies the resource, (b) identifies the media types of representations that the client is willing to consume in response (e.g., plain text or HTML; GIF or PNG), and (c) potentially is characterized as a form submission. The server responds with a resource representation that fulfills these constraints.

3. **Content Format** HTML [35] is a broadly implemented format for content, and started as a simple and declarative markup language. The early versions already included an anchor element which permits embedding hyperlinks to resources identified by URIs within the text. Based on plain text with embedded “tags”, this markup language can be written in a simple text editor, which is, remarkably, even done today, approximately 20 years later.

Notably, these three basic agreements are loosely coupled: While the URIs we use most frequently identify resources retrieved through HTTP, URIs can also be used to identify resources retrieved through other protocols, such as FTP, or even video chatting protocols like Apple’s Facetime [129]. HTTP can be used to retrieve URI-identifiable resources in just about any format that is represented in bits and bytes, such as HTML documents, images, PDF documents, or scripts (i.e., executable program segments generally written in an interpretable scripting language such as JavaScript). Similarly, the concept of hyperlinks exists in formats beyond HTML: PDF, and even Word documents might permit these. Nevertheless, the Web’s basic fabric is built on universal support for URIs, HTTP, and HTML.

The early Web was non-interactive, declarative, and stateless on the client side. While these properties, in all their simplicity, enabled the initial goals of the World Wide Web, they were insufficient to meet the demand for the rich application platform that the Web has become today. Without a doubt, JavaScript is the single most influential technology in the evolution of the Web. Initially intended to manipulate Web pages within the browser through the DOM, or Document Object Model (illustrated in Figure 1.1), JavaScript quickly proved much more powerful, making it the de facto client-side programming language of the Web. One of the driving factors behind the success of JavaScript is AJAX, a programming technique based on JavaScript and XML that enables asynchronous network operations. Using AJAX, Web pages are able to store and retrieve information in the background, integrating changes and new content into the Web page on the fly. Many modern Web applications still depend on

this technique, albeit that XML has been replaced with the JavaScript Object Notation (JSON), a JavaScript-based format for representing objects [44].

A second technological upgrade consists of the rich content types available in the Web today. Modern Web content is no longer limited to HTML and images, as modern browsers also support several audio and video formats, XML-based languages for defining images (SVG) [61] and scientific data (MathML) [51], and advanced styling information for HTML documents (CSS) [92].

Finally, the third essential component for a rich application platform is client-side state within the browser. A first approach encoded parameters in every URI, a very impractical solution, which was quickly replaced by cookies, which are key-value pairs issued by the server, stored by the browser and attached to each request. Today, cookies are used by practically every Web application, but have the disadvantage that they are attached to every request, making them unfit for storing large amounts of data. Therefore, several recently introduced APIs offer client-side storage in the form of key-value pairs [118], an object database [176] or a virtual file system [199].

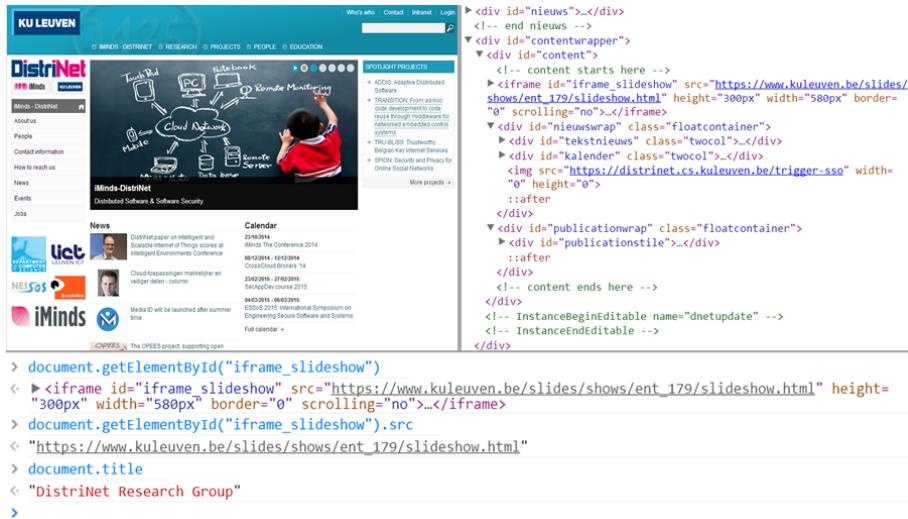


Figure 1.1: An illustration of a rendered HTML page (top left), its HTML code (top right) which is accessible from JavaScript (bottom) through the DOM interface.

These three major changes on top of the basic hypertext system have sparked the shift from a one-way information exchange to a bidirectional read/write Web, also known as Web 2.0. This new stage in the evolution of the Web

combines the technological advances with social aspects of actively participating users, resulting in dynamic applications, that actively improve as their number of users increases. Well-known examples are Wikipedia, Facebook and the many Google services we use throughout the day. The social effect even intensified when people started carrying always-on, always-connected devices everywhere they go. Smartphones enable instant and continuous access to information, further stimulating location and context-aware social services.

Further development of the browser towards an application platform has resulted in a paradigm shift, where more responsibilities are pushed towards the client. The client component is no longer simply a view on the application running in the backend, but has become the application, which interacts with a light, storage-centered backend application through rich, RESTful APIs. The technologies behind this paradigm shift are lightweight server-side technologies (e.g. NodeJS, Go), client-side libraries (e.g. JQuery, Bootstrap), and client-side templating frameworks (e.g. AngularJS, Ember.js). Additionally, browsers offer numerous useful features to Web applications, such as access to client-side storage facilities [176, 118], access to information about the device and its sensors [39, 151], and numerous communication mechanisms [116, 34].

This “appification” of the Web is further stimulated by the rise of mobile devices, with their restricted operating systems and vendor-controlled application stores. Not only is the majority of applications offered in today’s application stores based on Web technology [162], but recent standardization efforts [120] provide the necessary APIs to build Web applications that can interact with the underlying device, making them indistinguishable from native applications.

The Web has known several evolutionary steps, which have transformed the static server-side content into dynamic server-side applications, and have transformed dull page-viewing browsers into execution platforms running highly dynamic and powerful Web applications. We observe a similar trend in the evolution of Web security, resulting in a shift from server-side to client-side Web security.

## 1.2 The Relevance of Client-Side Web Security

The security landscape in the early Web was vastly different from what we see today. Attackers focused on server-side services, attempting to exploit the services or gain control over the server machine. Well-known examples of such attacks are SQL injection, command injection or the exploitation of buffer overflow vulnerabilities within server software. As the functionalities of Web services grew, attackers started targeting client machines, aiming at exploiting client-side vulnerabilities to install malware, for example to gain unauthorized

Table 1.1: The OWASP Top Ten Project [257] lists the most critical Web application security flaws. The gray colored rows indicate vulnerabilities that are relevant for client-side Web security.

1	Injection
2	Broken Authentication and Session Management
3	Cross-Site Scripting (XSS)
4	Insecure Direct Object References
5	Security Misconfiguration
6	Sensitive Data Exposure
7	Missing Function Level Access Control
8	Cross-Site Request Forgery
9	Using Components with Known Vulnerabilities
10	Unvalidated Redirects and Forwards

access to the victim’s bank accounts and other personal information. As memory corruption exploits in browsers or plugins became scarcer, the focus shifted more towards the “weaker” Web vulnerabilities. Attacks such as cross-site scripting and cross-site request forgery use the browser as a means to carry out actions on the server, in the name of the victim.

A perfect illustration of the attacker’s shift from server-side services towards the client side are two industry-driven surveys of the most important security vulnerabilities. Both the OWASP Top Ten Project [257] and the CWE/SANS Top 25 Most Dangerous Software Errors [172] include the typical server-side vulnerabilities, such as SQL injection and command injection, but also have allocated approximately one third of the slots to client-side security problems (shown in Table 1.1 and Table 1.2).

Naturally, when the attackers’ focus shifts towards the client, the counter-measures and security policies evolve as well. This evolution closely aligns with the evolutionary steps of the Web. The first security policies were static, encoded as default behavior in the browser, with the same rules for every Web application. Two examples of such static policies are the Same-Origin Policy, and the same-origin behavior of the XMLHttpRequest object, two policies which will be explained in the coming chapters. Next come the dynamic security policies, mainly enforced at the server side. Typical examples are token-based or request header-based protections against cross-site request forgery, or validation-based protections against cross-site scripting. These security policies line up with the rise of dynamic Web applications with a server-side processing component. The next wave of security mechanisms takes advantage of the browser transformation into an application platform. Modern security policies are enforced at the client side, but driven by the Web application at

Table 1.2: The CWE/SANS Top 25 Most Dangerous Software Errors [172] lists the most widespread and critical errors that lead to serious vulnerabilities. The gray colored rows indicate vulnerabilities that are relevant for client-side Web security.

1	Improper Neutralization of Special Elements used in an SQL Command
2	Improper Neutralization of Special Elements used in an OS Command
3	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
4	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
5	Missing Authentication for Critical Function
6	Missing Authorization
7	Use of Hard-coded Credentials
8	Missing Encryption of Sensitive Data
9	Unrestricted Upload of File with Dangerous Type
10	Reliance on Untrusted Inputs in a Security Decision
11	Execution with Unnecessary Privileges
12	Cross-Site Request Forgery (CSRF)
13	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
14	Download of Code Without Integrity Check
15	Incorrect Authorization
16	Inclusion of Functionality from Untrusted Control Sphere
17	Incorrect Permission Assignment for Critical Resource
18	Use of Potentially Dangerous Function
19	Use of a Broken or Risky Cryptographic Algorithm
20	Incorrect Calculation of Buffer Size
21	Improper Restriction of Excessive Authentication Attempts
22	URL Redirection to Untrusted Site ('Open Redirect')
23	Uncontrolled Format String
24	Integer Overflow or Wraparound
25	Use of a One-Way Hash without a Salt

the server side. One prime example is Content Security Policy [229], which is an application-specific policy enforced by the browser. Similarly, the recent work on Entry Points [206] proposes client-enforced protection against cross-site request forgery attacks. These server-driven, client-enforced policies are often used in a layered defense strategy, where both the client and the server enforce a security policy, striving to stop attackers, even when they manage to circumvent one of the security measures.

The content of this dissertation strongly affirms the evolution towards client-side security mechanisms. On one hand, we provide a broad overview of the client-side Web security landscape, which consists of several vulnerabilities that

have emerged from the fundamental technologies of the Web, and mitigation techniques that build upon the basic security policies of the Web. On the other hand, we take an in-depth look at session management, one specific building block of the Web which poses interesting security challenges, even until today. In four technical chapters, we propose client-side solutions to improve the security of Web sessions, supporting the evolution towards client-side security technologies. These client-side mitigation techniques push the autonomous client-side defenses to its limits, flirting with the delicate balance between usability and security.

### 1.3 Overview of this Dissertation

In this dissertation, we approach Web security from the client side, an important field of research that emerged since the mid 2000s. Besides a broad overview of client-side Web security, this dissertation makes several contributions towards improving the security of Web sessions and session management systems, with a strong focus on autonomous client-side mechanisms, that protect the user without cooperation of the vulnerable Web applications. We provide a sneak preview of these contributions below, but will revisit them in the conclusion (Chapter 8), at which point they will have been fully explained, allowing us to make a few concluding remarks.

In a first contribution (Chapter 2), we provide the necessary context to fully grasp the broadness of the field of client-side Web security. We discuss the relevant building blocks of modern Web applications and define the relevant threat models that govern the attacker's capabilities. As this dissertation focuses on session security problems in Web applications, we subsequently focus on several attacks impacting the client-side security, providing a detailed description of the attack, an overview of available mitigation techniques, state-of-the-art research and the current state-of-practice as observed on the Web. We converge towards the upcoming technical contributions by investigating specific threats against Web sessions, the main topic of this dissertation.

These four technical contributions each counter a specific threat against session management in Web applications. Implementations of each of these contributions illustrate the practical applicability, either as a mature product or as a proof-of-concept prototype.

- **CsFire** is a browser add-on that protects against cross-site request forgery (CSRF) attacks. CsFire autonomously decides when a cross-origin request is considered to be potentially harmful, and strips the cookies from these requests, rendering them harmless. The effectiveness of CsFire's request

filtering algorithm, and the associated *trusted delegation assumption*, is formally verified using the bounded model checker Alloy. Further, CsFire is publicly available for the Firefox and Chrome browsers, and has thousands of unique daily users. We present CsFire in Chapter 3.

- **Serene** is the first client-side mitigation technique against session fixation attacks, which give the attacker control over the user’s authenticated session. Serene is capable of protecting both cookie-based and parameter-based session management mechanisms. Serene is implemented as a browser add-on, and its effectiveness and compatibility evaluated on the Alexa top 1,000,000 sites. We present Serene in Chapter 4.
- **SecSess** fundamentally improves the security of Web sessions, as it upgrades the HTTP session management mechanism to prevent unauthorized transferring of the session. We have implemented the client-side part of SecSess as a browser add-on, and the server-side as a middleware for the Express framework on top of NodeJS. SecSess is fully compatible with current deployment scenarios on the Web, including the use of middleboxes throughout the network path, such as Web caches and perimeter security devices. We present SecSess in Chapter 5.
- **TabShots** is a browser add-on that detects tabnabbing attacks, a special variation of a phishing attack. TabShots visually compares screenshots of a browser tab, in order to detect potentially harmful changes. The detected changes are highlighted, thereby alerting the user when he wants to enter authentication credentials in a fraudulent form loaded by a tabnabbing attack. We present TabShots in Chapter 6.

Based on our experience gained during the inception and development of these practical countermeasures, we reflect on the use of client-side mitigation techniques by exploring the following topics in Chapter 7:

- **Experience Report on Client-side Mitigations.** Based on our experience with developing client-side mitigation techniques, we provide insights in the different implementation strategies, with their advantages and disadvantages. We further focus on browser add-ons, the preferred implementation strategy used throughout this dissertation, and the challenges we encountered during the evaluation of our prototypes.
- **Research Challenges and Trends.** We identify several research challenges that lie on the road ahead, and look into trends that can be observed in the current state-of-practice. Concretely, we elaborate on the importance of theoretical approaches towards Web security, the relevance of upcoming state-of-practice security policies that can be used

as a second line of defense, and the rise of Web technologies as an essential building block of mobile apps.

We conclude the dissertation (Chapter 8) with a brief summary of the previous chapters, and revisit our contributions in the full context of this dissertation.

## Chapter 2

# Background on Client-Side Web Security

As explained in the previous chapter, the remainder of this text will cover client-side Web security, which lines up with the recent evolution towards client-side security measures. This chapter introduces the relevant background knowledge to establish a common understanding of Web applications, threat models and common client-side attacks against Web applications. The information presented in this chapter is scoped specifically towards client-side Web security, with a focus on the security of Web sessions. Our primer on client-side Web security offers a broader view on client-side Web security [67].

The content covered in this chapter is scoped towards Web technologies on the application layer [226], and effects from underlying protocols that seep through in Web technologies, such as the *Secure* attribute for cookies sent over a TLS-secured connection. The security of the underlying protocols is considered to be out of scope, for example cryptographic attacks on the TLS protocol. Since this dissertation focuses on client-side Web security, specific server-side technologies and infrastructure are also considered to be out of scope, unless relevant for client-side countermeasures. A similar argument holds for networking infrastructure. This chapter concludes with the introduction of three concrete threats against Web sessions and session management mechanisms: (i) violating the integrity of a session, (ii) unauthorized transfer of a session, and (iii) impersonating a user by establishing a new session. The practical contributions that will be presented in the next four chapters each address one of these threats, which is why this chapter is used to provide an adequate frame of reference.

## 2.1 Building Blocks of Web Applications

Modern Web applications are built in a highly dynamic and complex environment, consisting of numerous interacting components and security policies, but also subtleties and potential security pitfalls. This section covers the building blocks of a Web application that are relevant for client-side Web security, and introduces them by means of an example application. Figure 2.1a shows a screenshot of our insecure banking application, which has been instrumental to demonstrate CsFire, our add-on that protects against cross-site request forgery attacks, at numerous local and international events, including OWASP conferences and chapter meetings, iMinds the Conference, and departmental events within the university.

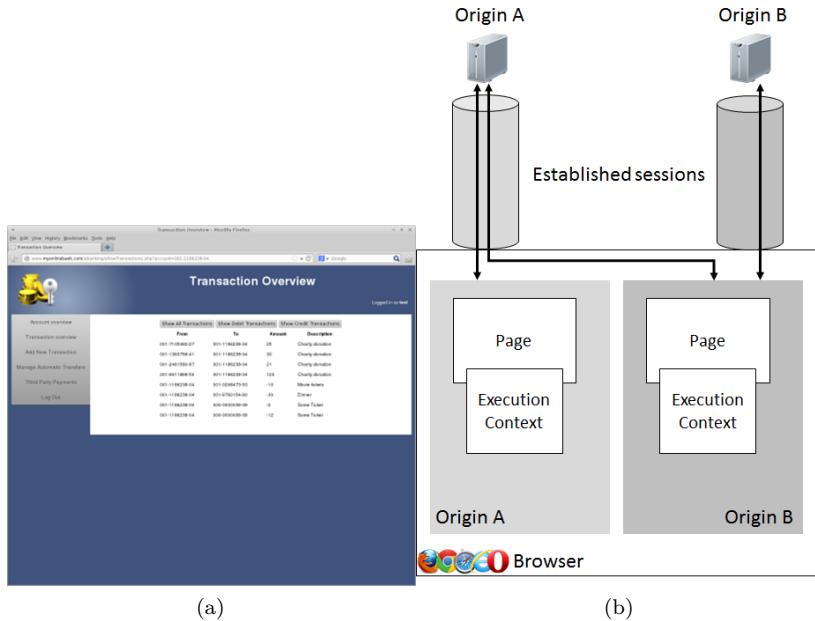


Figure 2.1: A screenshot (a) and conceptual view (b) of the insecure banking application, often used to demonstrate CsFire. Already note that in the right figure, a context with origin B communicates with origin A using an already-established session, a source for many Web security problems.

The screenshot of the banking application shows a straightforward Web application, but under the hood, numerous components and browser features enable a dynamic, interactive and complex Web application. The components

depicted in Figure 2.1b are only a limited selection of the most relevant features, but numerous others make the Web security landscape even more complex [66, 67]. Examples are browser add-ons, content plugins, background JavaScript threads [117], JavaScript’s cryptographic APIs [60], the ability to register protocol and content handlers [35], etc.

Within the browser, the *window* is a container object holding all the Web application’s data, such as the currently displayed page and the associated JavaScript executing context. Additionally, the *window* offers access to the browser-provided features, such as the numerous APIs, the cookie jar, etc. While even a single *window* container already seems to be a complex structure, modern browsers not only support multiple *window* objects to live simultaneously next to one another as tabs or browser windows, they also support the dynamic nesting of *window* containers using the HTML *frame* or *iframe* tags. The specific features of these components and their interactions will be covered in more detail below.

### 2.1.1 Client-Side Execution Context

The client-side execution context controls the dynamic behavior of a Web application within the browser. The execution context runs the application’s JavaScript code, which has access to the page’s HTML tree and contents through the Document Object Model (DOM) [250], as well as to the browser-provided features and APIs, which enable navigation of the page, remote communication, interacting with the client device, and storing information at the client side. A client-side execution context has a specific origin, known as the triple *(scheme, host, port)*, which is derived from the URI of the page associated with the execution context.

As the Web matured, technologies like frames, tabs, and pop-up windows permitted the simultaneous execution of several Web applications in their respective client-side execution contexts. These contexts can to some degree communicate with one another within the browser, as well as with remote servers. The browser’s core security policy, the Same-Origin Policy (SOP), enforces restrictions on this communication, to prevent execution contexts with different origins from directly influencing one another. The SOP introduces a basic security boundary that prevents a resource from accessing another resource’s context, unless its origins match. As a result, different Web applications can coexist in the same Web browser with a basic isolation and confidentiality guarantee against one another.

With the introduction of additional browser features, these restrictions imposed by the Same-Origin Policy have become even more important. For example,

the modern client-side storage APIs [118, 176, 199] use origin-specific storage containers, restricting access to the specific container associated with the requesting script’s context. Similarly, several APIs require explicit user permission before exposing sensitive data or features towards the execution context, and these permissions are associated with the requesting context’s origin. Examples are sharing the device’s location [196] and capturing audio or video [46].

While the SOP effectively separates execution contexts from different origins from one another, there is no separation mechanism that supports the isolation of a specific piece of code within an execution context. The main use case for such code isolation is the inclusion of remote JavaScript files, used by almost every modern Web application to include libraries, advertisements or analytics code [182]. As this third-party code is integrated into the host page’s execution context without any boundaries, the third-party code gains access to all the features associated within the host page’s origin. This effectively imposes a strong trust relationship between the host origin and the third-party provider, which is easily violated through malicious behavior, or by an attacker that compromises the third-party provider [138].

### 2.1.2 Session Management

In the early Web, when sites only consisted of static content, the HTTP protocol already offered an *Authorization* header [99], which could be used for authentication and authorization, even up until this day. The most common application of the *Authorization* header uses *Basic* authentication, where the user’s credentials, more specifically a username and password, are base64-encoded, and included as the header value. This allows a server-side application to extract these credentials, verify them against a password file and make a decision on whether to allow the request or not. After a successful authentication, the browser will attach the user’s credentials to every subsequent request to this origin. Note that since the browser remembers these credentials during the lifetime of its process, logging out of an account is only possible by closing the browser.

As Web applications became more complex, developers wanted to integrate authentication with the application, streamlining the user experience within the same look and feel. To authenticate users, they embedded an HTML form, where the user had to enter a username and a password. By submitting the form, the username and password were sent to the server, where they could be validated. However, since the credentials are only sent in a single request after form submission, instead of in every subsequent request as with

the *Authorization* header, Web applications needed a way to keep track of the user's authentication state across requests, a challenge with the stateless HTTP protocol.

A session management mechanism on top of HTTP is capable of associating multiple requests from the same user with a server-side session state, allowing the application to store useful session information, such as an authentication state or a shopping cart (illustrated in Figure 2.2). To keep track of this server-side session across multiple requests, the session is assigned an identifier, which the client needs to include in every request. Initially, this identifier was embedded in the URI as a parameter, for example like `http://example.com/index.html?SID=1234`. Unfortunately, this requires the Web application to append the user's session identifier to every URI in the page, and repeat this action for every user. The introduction of cookies [23] addresses this problem. Cookies are server-provided key-value pairs, stored by the client in the cookie jar and attached to every request to the same domain. Essentially, cookies allow the server to store small pieces of data at the client side, which will be attached to future requests. Cookies can be used for storing simple settings, such as a language preference, or for building more complex systems, such as session management mechanisms. Today, cookie-based session management is the de facto standard, but many systems still allow parameter-based session management as a fallback mechanism, for example when cookies are not supported by the browser, or disabled by the user.

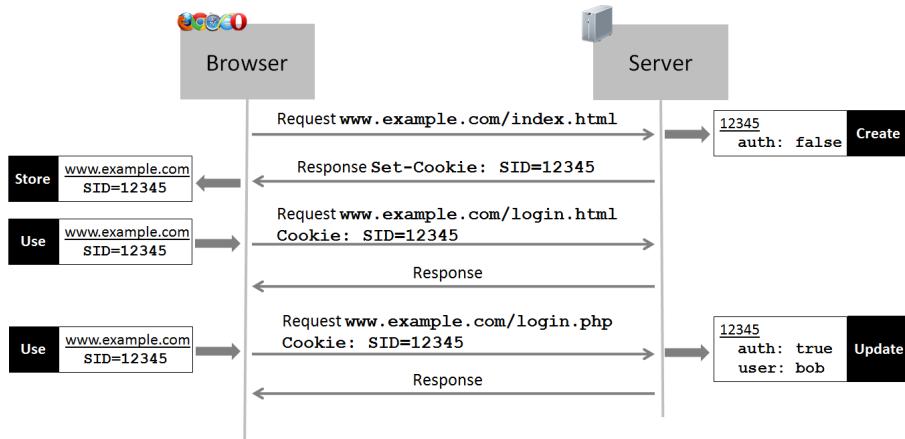


Figure 2.2: An illustration of cookie-based session management on top of HTTP. Presenting the session identifier (in this example 12345) to the server suffices to associate a request with the server-side session object.

In this de facto standard session management mechanism, the session identifier has a peculiar property, since it suffices to present the associated session identifier to the Web application in order to associate a request with a specific session. This essentially makes the session identifier a *bearer token*, meaning that a request *bearing* the identifier is granted the privileges associated with the session. Since the server-side session state generally stores the user’s authentication state, these privileges are comparable to those of an authenticated user. Note that this means that the session identifier and credentials are similar, as they both grant access to an authenticated session. Credentials, however, can be used to repeatedly gain access to the application, while the level of access granted by the session identifier is limited to the lifespan of the session.

### 2.1.3 Cross-Origin Interactions

Even though the Same-Origin Policy effectively prevents execution contexts from different origins from directly influencing one another, several interaction patterns are present in the Web, or are explicitly enabled. A first example is local interaction between different execution contexts, a feature highly-demanded by mashup developers [63]. Today, the Web Messaging [115] specification offers an opt-in mechanism to enable controlled interaction between different execution contexts, and is currently available in all modern browsers. Such controlled interaction allows different contexts to cooperate or exchange information, which in turn enables alternative architectures that isolate security-sensitive components in their own origin [11, 177].

A second example consists of an implicit interaction when a page from origin A makes a request to a server located within origin B, for example to include a script or an image from origin B. When the browser sends out such requests, it will attach any cookies that it has stored for origin B, allowing the cross-origin request to become part of the user’s session with origin B. This interaction pattern is a crucial requirement for numerous interactions in the Web, such as Facebook’s *like button*, which can be embedded in arbitrary pages, allowing users to like the page on their Facebook accounts by simply clicking the button. Unfortunately, as will be covered in the coming sections, this interaction pattern also lies at the basis of Cross-Site Request Forgery (CSRF) attacks.

A final example of cross-origin interactions are the intricacies of explicit interactions initiated by a script running in a page’s execution context. The traditional script-based communication mechanism uses the XMLHttpRequest object to send custom requests, and was restricted to communication within the same origin. This limitation has sparked creative solutions to bypass the same-origin restriction using script tags, JSON [44] and “padding”. This technique,

called JSON-P, dynamically includes additional JavaScript files, which will invoke a callback function in the local execution context. This effectively enables cross-origin communication, since the browser can send parameters in the request for a script file, and the server responds with the requested data. Unfortunately, this also introduces a severe security vulnerability, since the server can willingly or unwillingly inject content into the page's execution context.

In response to this dangerous practice, the XMLHttpRequest Level 2 specification [249] makes cross-origin communication explicitly possible by implementing the Cross-Origin Resource Sharing (CORS) specification [248]. CORS allows servers to provide the browser with a security policy that explicitly allows certain cross-origin interactions to be sent from a local execution context. As CORS is an opt-in policy, legacy servers that are not aware of these new capabilities will not become vulnerable to new attacks. While CORS largely succeeds in this effort, some caveats still remain [66].

### 2.1.4 Requests and Responses on the Network

Naturally, as the Web is a distributed system, it depends on an underlying network infrastructure and the associated communication protocols. Even though the details of HTTP [99], the de facto communication protocol of the Web, are well-encapsulated, it still introduces dependencies and peculiarities that influence the Web's security model in subtle ways.

A first effect of the use of HTTP and the associated *http* scheme, comes from the plaintext nature of the protocol. Without additional protections, all data sent to an origin with the *http* scheme is unprotected once it leaves the browser, enabling network-based attacks, such as eavesdropping or man-in-the-middle attacks. Such attacks can have serious consequences, as user credentials, session identifiers and personal information is often transmitted from the browser to the server. In response to this insecure practice, HTTPS was introduced, which essentially means running the HTTP protocol over a channel secured by Transport Layer Security (TLS), earlier known as Secure Sockets Layer (SSL). Such a channel offers entity authentication, confidentiality and data authenticity on the transport level, thereby preventing passive or active network-level attacks. Even though a TLS connection is initiated at the network level, its effects do influence several important Web concepts, such as the *Secure* flag for cookies.

A common deployment scenario of HTTPS is to offer the application's main content over HTTP, but to switch to HTTPS for submitting sensitive information, such as authentication credentials or payment information. Often, the application reverts back to HTTP once the sensitive information has been

submitted, generally sharing the same session for both the HTTP and the HTTPS traffic. Such a deployment scenario subjects the plaintext HTTP traffic to network attackers, leading to attacks such as session hijacking.

A second example of how HTTP and HTTPS are difficult to combine is known as *mixed content* [53]. Such a scenario occurs when a page is loaded over a secure HTTPS connection, which effectively prevents in-transit tampering on the network, and the page subsequently loads additional resources, such as JavaScript files, over a plaintext HTTP connection. Since these resources can be tampered with on the network level, an attacker can succeed into injecting code in the application’s execution context, allowing him to take control over the application running in the user’s browser.

## 2.2 Threat Models

The previous section introduced several building blocks of modern Web applications, and already hinted at several of the attacks that are possible in the current Web. This section presents several threat models, often encountered in research on Web security, and in this case, relevant for the discussion of client-side Web security. The general capabilities for each threat model are explained below, and additional details are provided in the upcoming chapters, where the threat models are tailored towards a specific problem domain.

**Forum Poster.** A *forum poster* [26] is the weakest threat model, representing a user of an existing Web application, who does not register domains or host application content. A forum poster uses a Web application, and potentially posts active content to the application, within the provided features. Additionally, a forum poster remains standards-compliant, and cannot create HTTP(S) requests other than those he can trigger from his browser.

**Web Attacker.** The *Web attacker* [10, 26, 27, 41] is the most common threat model encountered in papers, and represents a typical attacker who is able to register domains, obtain valid certificates for these domains, host content, use other Web applications to post content to, etc. Since none of these capabilities requires a specific attacker characteristic, such as a specific physical location, every user on the Web is able to become a Web attacker. As a consequence, the capabilities of the Web attacker are considered to be the baseline for threat models in the Web, with the forum poster as the exception to the rule.

**Gadget Attacker.** A *gadget attacker* [10, 27] is a more powerful variant of the Web attacker, where the attacker hosts a component that is wilfully integrated into the target application. Popular examples are JavaScript libraries, such as JQuery, analytics code, such as Google analytics, or widgets, such as Google Maps. The gadget attacker is extremely relevant in the context of code isolation for mashups or complex, composed sites, which integrate content from multiple stakeholders with varying trust levels.

**Related-Domain Attacker.** A *related-domain attacker* [41] is an extension of the Web attacker, where the attacker is able to host content in a related domain of the target application. By hosting content in such a related domain, the attacker is able to abuse certain Web features, which are bound to the parent domain. This is the case when the attacker is able to host content on a sibling or child domain of the target application, for example for the Web sites of different departments within a company.

**Related-Path Attacker.** A *related-path attacker* is another extension of the Web attacker, and represents an attacker who hosts an application on a different path than the target application, but within the same origin. This scenario occurs for example within the Web hosting of Internet Service Providers (ISPs), which often offer each of their clients a Web space under a specific path, all within the same origin. Academic papers aptly describe this attacker [136] and its conflicts with the Web’s security model, albeit without giving it an explicit name.

**Passive Network Attacker.** A *passive network attacker* [137] is considered to be an attacker who is able to passively eavesdrop on network traffic, but cannot manipulate or spoof traffic. A passive network attacker is expected to learn all unencrypted information. Additionally, a passive network attacker can also act as a Web attacker, for which no specific requirements are needed. Depending on the system under attack, the passive network attacker may require a specific physical location to eavesdrop on the network traffic. One common example of a passive network attack is an attacker eavesdropping on unprotected wireless communications, which are ubiquitous thanks to publicly accessible wifi networks and freely available hotspots.

In 2013, whistleblower Edward Snowden [239] revealed that intelligence services across the globe have powerful traffic monitoring capabilities. These *pervasive monitoring* capabilities are essentially passive network attacks, albeit on a very large scale compared to the traditional passive network attacker. In response to

the Snowden revelations, the IETF has drawn up a best practice, stating that specifications should account for pervasive monitoring as an attack [97].

**Active Network Attacker.** An *active network attacker* [10, 26, 137] is considered to be capable of launching active attacks on a network, for example by controlling a DNS server, spoofing network frames, offering a rogue access point, etc. An active network attacker has the ability to read, control, inject and block the contents of all unencrypted network traffic. An active network attacker is generally not considered to be capable of presenting valid certificates for HTTPS sites that are not under his control, unless by means of obtaining fraudulent certificates, or by using attacks such as SSL stripping [171].

## 2.3 Common Client-Side Attacks

Based on the threat models defined in the previous sections, attackers can carry out several attacks within the Web platform. Some of these attacks originate from implementation vulnerabilities, while others are inherent to the design of the Web, and are indistinguishable from legitimate interaction patterns in the Web.

This section covers eight common client-side attacks, which are relevant for the remainder of this dissertation. The attacks are ordered in such a way that they come gradually closer to the user, starting from within the network, followed by a simultaneous session in the browser, taking control of an existing session, to end with personal attacks on the user. The description of each of the attacks covers the problem and its roots, currently available mitigation techniques, state-of-the-art research and the current state-of-practice.

The information on the current state-of-practice is both timely and relevant, as it is based on the results of a crawl performed in June 2014 on the Alexa top 10,000 sites, in total good for 4,185,227 requests.<sup>1</sup> The crawl data is analyzed for deployments of well-known and recently introduced mitigation techniques, giving an up-to-date view on the adoption rate of certain mitigation techniques. This shows even the most recent security technologies are already being adopted across the Web.

---

<sup>1</sup>My colleague, Steven Van Acker, deserves the credit for crawling the Web and providing the raw data for our analysis.

### 2.3.1 Eavesdropping Attacks

In an eavesdropping attack, a *passive* or *active network attacker* listens in on other users' network traffic, such as DNS queries, HTTP requests and responses, etc. By eavesdropping on their network traffic, an attacker is not only able to learn sensitive, personal information, such as credit card info, financial means, usernames, passwords, contents of email messages, etc., but can also listen in on important Web metadata, such as session identifiers or supposedly secret cookies. Obtaining any of this information is not only directly harmful to the user, but also enables the attacker to escalate the attack, for example through *session hijacking*.

Eavesdropping attacks are extremely relevant in the modern Web, especially because of the numerous wireless networks, to which users connect with their mobile devices or laptops. Many of these networks are unprotected or easily spoofed by an attacker. Additionally, with the revelations of Snowden [239], it has become clear that state-sponsored eavesdropping occurs on a large scale, scooping up every piece of unencrypted information that is encountered.

#### Description

The goal of an eavesdropping attack is to obtain traffic that is sent over the network. The way of executing an eavesdropping attack depends on the network under attack. For example, eavesdropping on airborne signals, such as wifi, radio or cellular, only requires an antenna in the proximity of the network. Eavesdropping on a switched, wired network requires some interference, for example by running an ARP spoofing attack. Eavesdropping can also occur at intermediaries within the network infrastructure, for example at an ISP, a proxy server or a Tor [82] exit node. Even higher up in the network, an attacker can eavesdrop on backbone traffic, with submarine taps on fiber-optic cables [19] as an extreme example.

Technically, running an eavesdropping attack is fairly straightforward. As an illustration, the browser add-on *Firesheep* [48] enables a user to eavesdrop on a wifi network, abusing obtained session identifiers to perform a session hijacking attack with one point-and-click operation. Alternatively, software tools such as *Subterfuge* [243] and dedicated devices such as the *Pineapple* [111] make collecting sensitive information a straightforward task. Eavesdropping on a wired, switched network is also possible with a wide variety of freely available tools, such as *Ettercap* [93] or *dsniff* [224].

Essentially, eavesdropping attacks will always be possible, especially with the evolution towards wireless networks. However, the real problem with

eavesdropping is the huge amount of information that is transmitted in the clear, making the physical access to the network signals the only barrier to overcome.

## Mitigation Techniques

The main approach to protect network traffic against eavesdropping attacks is to deploy security protocols, effectively protecting the data being sent over the network. The use of network-specific data link-layer security protocols, such as WPA2 [256] and EAP [3] can effectively help in mitigating a local eavesdropping attacker, but does not protect the traffic against eavesdropping beyond the local network.

An approach offering end-to-end security is using HTTP deployed over TLS [80], where TLS is aimed at offering confidentiality, data authenticity and entity authentication. The confidentiality property effectively eliminates the usefulness of the data in transit to an eavesdropping attacker. As the next section will explain, the data authenticity and entity authentication properties mitigate active network attacks. Note that TLS uses certified public keys to establish entity authentication, but that confidentiality against passive network attackers can already be achieved using self-certified keys. Such a configuration is however not recommended for general use, due to the weaker security guarantees.

The TLS protocol itself is undergoing constant revision and its security is the topic of ongoing cryptographic research. Recent examples are the discovery of attacks allowing the extraction of cookies from an encrypted stream [15], and the identification of weaknesses in the RC4 algorithm [14], supporting the common belief that RC4 should be considered broken. Reacting to these results and other similar research output, the TLS working group within the IETF is currently considering new cipher suites [155].

In addition to new ciphers, TLS deployment is also an important factor to consider. Older versions of TLS remain in use for a considerable period, even after the introduction of newer versions with better security features. For example the attacks mentioned in the previous paragraph have already been countered via authenticated encryption cipher suites that were defined as part of TLS 1.2 [79], but so far version 1.2 has not seen very widespread deployment. However, in response to these attacks, deployment roadmaps for TLS 1.2 have been accelerated by browser vendors and the open-source security community.

Another standardization proposal focuses on a new Best Current Practice (BCP) for the use of TLS on the Web [221], aiming to aid Web application developers and administrators in the use of TLS. One example strategy that is advocated

is the achievement of *perfect forward secrecy* (PFS), which guarantees that previously recorded TLS sessions cannot be deciphered by learning the server's private key at a later point in time. While PFS has previously known a limited deployment due to an impact on performance and requiring less commonly used cipher suites, it has come into the spotlight again, due to private keys leaking out, for example if someone hacks a Web server, or if server units are decommissioned inappropriately.

Finally, a large effort is being spent on the development of the successor to HTTP, HTTP/2.0 [32] based on Google's SPDY protocol [31]. While initially SPDY was proposed to always run over TLS, that position was somewhat watered down in HTTP/2.0, mainly due to interference with middleboxes, such as Web caches or HTTP proxies that need to be able to see some HTTP metadata (e.g., headers) to function. Nonetheless, HTTP/2.0 will be far more likely to be deployed running over TLS, since the *application layer protocol negotiation* TLS extension [103] offers the most efficient upgrade path, with the fewest additional network round trips. Additionally, discussions to allow clients and servers to make use of TLS even for HTTP (i.e. non-HTTPS) URIs when using HTTP/2.0, are ongoing [188].

## State-of-Practice

While TLS effectively tackles network attacks, it is not yet widely deployed, although adoption is growing. As an indication of the current deployment state of TLS, a monitoring site [251] reports that approximately 34% of the top 10-million Web sites are using TLS with certificates issued by a recognized CA. Recently, Google has announced an *HTTPS Everywhere* initiative, encouraging the deployment of TLS. As part of this initiative, Google is starting to use HTTPS as a signal in their ranking algorithm [20].

Next to the limited adoption, older and less secure versions of TLS that are deployed, are rarely upgraded to the latest version, leaving a trail of inadequate legacy implementations across the Web. The SSL Pulse project [198] reports that of the 152,733 surveyed TLS sites in July 2014, only 28.3% had a secure TLS deployment.

While the specific reasons for the slow adoption of TLS are hard to pinpoint, potential candidates are its (antiquated [153]) reputation imputing significant performance impact, the difficulty of managing and deploying certificates, potential interference or incompatibility of the encrypted traffic with middleboxes, such as proxies and caches, and general ignorance from Web application operators. Additionally, TLS is often used incorrectly, which may be attributed to relatively hard-to-use APIs and incorrectly configured trust-roots.

### 2.3.2 Man-in-the-Middle Attacks

In a man-in-the-middle attack (MitM), an *active network attacker* positions himself in the network, between the victim and the targeted Web application. This position not only allows the attacker to inspect all traffic that is sent between the victim and the target application, but also allows modification of the traffic. Such a compromise gives the attacker full control over the user's actions, with potentially disastrous effects. Note that there are also "legitimate" use cases for performing a MitM attack, such as ISPs injecting advertisements into HTTP responses, or corporations deploying a Web content filter, responsible for filtering unwanted or harmful content.

MitM attacks are more sophisticated than eavesdropping attacks, but also occur frequently on the Web. Little is known about MitM attacks being carried out by small-scale attackers, but they do occur on larger scales, such as for state-sponsored censorship as seen in the Middle East, China, etc. Similarly, the same technology is used for scenarios where user consent is given, such as companies that deploy content filtering on their own networks, as a perimeter security measure [127]. Another known case is the NSA's QUANTUM INSERT program, where a MitM attack is used as an attack vector to install malware on the victim's machine [239].

#### Description

The goal of a man-in-the-middle attack is to be able to inspect and manipulate the victim's network traffic. This allows an attacker to modify legitimate transactions, carry out actions in the user's name, compromise files that are being sent to and from the victim, etc. TLS-secured connections with validated certificates are designed to withstand man-in-the-middle attacks, but flaws in the supporting systems may allow for subtle attacks to be carried out anyway. These flaws are caused by misplacement of trust in certain parties, or by placing the decision-making burden on the user.

Becoming a man in the middle in the network can be achieved at many levels. An attacker can physically place a machine in the network path, forcing the data to flow through this machine, or he can manipulate the network's parameters, to act as a gateway at the logical level, for example through ARP poisoning attacks. While the technical details on becoming a man in the middle are less important in this context, the impact of a MitM attack on the Web cannot be discarded. Once an attacker positions himself in the middle, inspecting and manipulating traffic becomes straightforward. One common example is in *mixed content* deployments [53], where an attacker can manipulate HTTP content

that is included by an HTTPS page, resulting in a compromise of the page's execution context.

Traditionally, TLS is deployed to prevent eavesdropping and MitM network attacks, since it offers confidentiality, integrity and entity authentication. The confidentiality and integrity effectively prevent an attacker from reading and modifying any network traffic, while the entity authentication property ensures that the involved parties are who they claim they are, thereby preventing a MitM attack within the TLS connection. Even though TLS is designed to counter MitM attacks, in reality, they remain possible for several reasons.

In 2009, Moxie Marlinspike argued [170] that users visiting a secure Web application probably will not type the *https://* part of the URI manually, meaning that the initial request will be made over HTTP. Typically, Web applications then redirect the user towards the correct HTTPS URI, causing a transition from HTTP to HTTPS. Exactly this transition can be exploited by an attacker who sits between the victim and the target application, causing the downgrade of the connection from HTTPS to HTTP, which is called an *SSL Stripping* attack [171].

Second, the entity authentication in TLS is based on private/public key pairs, of which the public key is verified by a Certificate Authority (CA), which is part of the Public Key Infrastructure (PKI). Unfortunately, the trust model in this scenario consists of the union of all public keys of CAs registered in the browser. Hence, any CA in the Web's PKI can issue a certificate for any Web site, in spite of the availability of the technology to constrain the trust in a specific CA [56]. With approximately 1,482 CAs trusted by Microsoft and Mozilla [87], any Web site is vulnerable to an attack with fraudulent but verified certificates being issued. To make matters even more complicated, browsers have a history of not consistently checking the revocation status of certificates using the Online Certificate Status Protocol (OCSP) [154], and alternatively propose a static list of revoked certificates. Even if they check the status with OCSP, the absence of an answer is ignored, allowing an active network attacker to circumvent the OCSP check.

Third, whenever an invalid certificate is encountered by a browser, the burden of the security decision is placed on the user. Regardless of whether the invalid certificate is caused by an expired expiration date, or a complete mismatch with the targeted Web site, browsers show scary warnings, asking the user to decide whether to trust the site or not. Since users also encounter these warnings for legitimate sites, a simplistic MitM using an invalid certificate has some chance of success.

A fourth degradation of the CA system in TLS comes from the deliberate MitM

devices, deployed by enterprises and large organizations with the goal of filtering inbound and outbound Web traffic. Reasons to deploy such filtering mechanisms go from offering protection, for example with a Web application firewall (WAF), to preventing employees from accessing sites that are deemed inappropriate, such as social networking applications. The problem with such devices is that in order to perform a MitM function on secure connections, they either have to install their own certificate on a user’s machine, or they have to obtain a valid certificate for every TLS-protected Web site on the Web. The former is a configuration hassle, which only works if you control all the client-side devices as well, and the latter seems impossible. Unfortunately, the system does not prevent collaboration between CAs and vendors of MitM devices [240], thereby harming the trust placed in the system.

Finally, the trust placed in CAs is easily abused when a CA is compromised. For example, the hacking of DigiNotar [197] resulted in the issuing of fraudulent certificates, allowing MitM attacks on secure connections to Yahoo, Mozilla, WordPress and the Tor project. The trusted roles of CAs can even be further compromised by government coercion to issue fraudulent certificates. This strategy is believed to be common practice in non-democratic countries [218], but recent revelations seem to implicate that this practice is widely deployed by secret agencies across the world [174].

The essence of the problem with MitM attacks, especially against TLS connections, is the misplaced trust in the system on the one hand, and the burden of the security decision on the user on the other hand. Clearly, blindly including every certificate of a root CA on the Web in the browser has been proven to be a bad idea, and typical Web users are not capable of making technical decisions about trusting a certificate or not.

## Mitigation Techniques

For long, the main mitigation technique for SSL stripping attacks has put the burden on the user, who should detect the presence of the lock icon to indicate a secure connection. One technological solution is provided by the HTTPS Everywhere [89] browser add-on, which forces the use of HTTPS on sites that support it. By forcing the use of HTTPS, SSL stripping attacks are effectively mitigated, since a direct HTTPS connection will be made. The research proposal *HProxy* [187] prevents SSL stripping attacks by leveraging the browser’s history to compose a security profile for each site, and validating any future connections to the stored security profiles. This approach effectively detects and prevents SSL stripping attacks without server-side support and without relying on third-party services. Finally, the ForceHTTPS research

proposal [137] has resulted in *HTTP Strict Transport Security (HSTS)* [121], which allows a server to require that browsers supporting HSTS can only connect over HTTPS, effectively thwarting any SSL stripping attacks. A server can enable HSTS protection by including a **Strict-Transport-Security** response header, declaring the desired lifetime for the HSTS protection. One caveat to HSTS being implemented as a response header, is the first contact with a site, when it is unknown whether an HSTS policy applies or not. This issue has been addressed by modern browsers including a predefined list of HSTS-enabled sites, effectively avoiding an initial HTTP connection.

Mitigation techniques against MitM attacks on TLS focus on determining the trustworthiness of the presented certificate. Certificate Transparency (CT) [156] aims to maintain a public, write-only log of issued certificates so that either user agents or auditors can detect fraudulent certificates. This would require a user agent to query the log during the TLS handshake, and auditors can query the log offline, to check for certificates being unexpectedly issued for one of their sites.

A second approach is based on detecting discrepancies between the currently presented certificate, and previously seen certificates, a technique called *certificate pinning* or *public key pinning*. While this approach requires the first connection to be secure, it effectively enables the detection of unexpected future updates. This approach is implemented in the Certificate Patrol browser add-on [179], and proposals to achieve this at the HTTP, TLS or other layers have been made [94, 121]. Note that public key pinning does not require a CA-signed certificate, and is compatible with self-signed certificates. Alternatively, Google has taken the approach of hardcoding the certificate fingerprints of Google-related TLS certificates, allowing Google Chrome to detect a potential MitM attack, even with a fraudulent certificate issued by a CA. Naturally, controlling both the services and the client platform is a key to the success of this approach.

Several proposals for alternate schemes to verify certificates have been made and evaluated [108], but the standardization work on Certificate Transparency seems to be most likely to gain widespread support, which is required for it to become an effective mitigation technique. In addition to CT-like approaches, DNS-based Authentication of Named Entities (DANE) [122] leverages the security of DNSSEC to bind certificates to domain names. DANE is perhaps less suited for the Web due to the current lack of deployment of DNSSEC and a corresponding lack of a well-defined transition path from today's PKI to a DANE-based PKI.

## State-of-Practice

Currently, a large part of the Web still transfers content over HTTP, making a man-in-the-middle attack trivial. In the last few years, major sites have started to switch TLS on by default, which has even increased after the revelations about pervasive monitoring. Adoption of HSTS is still in its early stages, but our July 2014 survey of the Alexa top 10,000 domains shows that 388 already send an HSTS header. Similarly, Chromium’s predefined list of HSTS-enabled sites counts 438 entries.

A recent study by Huang et al. [127] has discovered that forged certificates do occur in the wild. Of the 3,447,719 real-world TLS connections, at least 6,845 (0.2%) used a forged certificate. The authors attribute these forged certificates to adware, malware and security tools such as antivirus software, parental controls and firewalls.

Finally, in a recent security push by browser vendors, active mixed content is being blocked, preventing a compromise of the execution context through a network-level attack. While modern desktop browsers effectively enforce this policy, mobile browsers still include mixed content without constraints.

### 2.3.3 Cross-Site Request Forgery

A Cross-Site Request Forgery (CSRF) attack enables an attacker to forge requests to the target application from a legitimate user’s browser. A vulnerable application handles these forged requests the same way as legitimate requests from the victim. Successful CSRF attacks can trigger many actions in vulnerable applications, such as modifying account settings, or stealing money through an online banking system [264].

CSRF is prevalent in modern Web sites, and is ranked in both the OWASP Top 10 project [257] and the CWE/SANS Top 25 Most Dangerous Programming Errors [172]. Both small and large scale projects are affected, with for example CSRF vulnerabilities in online banking systems [264], Gmail [114] and eBay [149].

#### Description

The goal of a CSRF attack is to forge a request from a victim’s browser to the target application, triggering state-changing effects in the target application.

Examples of such state-changing effects are modifying account settings or adding items to a shopping cart. For a CSRF attack to succeed, it is essential that the user is already authenticated to the target application. From the application's point of view, the forged request has the same structure as a legitimate request, and is therefore indistinguishable from a legitimate request.

Tricking the victim's browser into making a request to the target application is a straightforward task. Browsers frequently issue requests to numerous unrelated sites, for instance when loading external resources such as images, stylesheets or a document to load in a frame, but also when submitting form data to a cross-origin URI. An attacker can easily include code that triggers a request to the target application in his own Web site or a site he controls. Alternatively, he can inject malicious HTML or JavaScript code in an unrelated but legitimate site, for example by posting code in the comments section of a legitimate forum. These two attack vectors respectively require the capabilities of a *Web attacker* and *forum poster*. Listing 2.1 shows the code that uses a hidden image to trigger a forged request, and Listing 2.2 shows a cross-origin form submission.

---

```

1 

```

---

Listing 2.1: A CSRF attack carried out by a hidden *img* tag that triggers a GET request to the target application.

---

```

1 document.getElementById("somediv").innerHTML += "<iframe
2         id='attackframe' style='height: 0px; width: 0px;'></iframe>";
3 var f = document.getElementById("attackframe");
4
4 var code = "<form id='attackform' action='http://admin.example.com/
5             createAccount.php' method='POST'>";
6 code +=     "<input type='hidden' name='username' value='attacker'>";
7 code +=     "<input type='hidden' name='password' value='12345678'>";
8 code +=     "<input type='hidden' name='action' value='create'>";
9 code +=   "</form>";
10
11 f.contentDocument.body.innerHTML = code;
11 f.contentDocument.getElementById("attackform").submit();

```

---

Listing 2.2: A CSRF attack carried out by JavaScript code that creates a hidden *iframe* containing a *form*, which is then automatically submitted to the target application at *admin.example.com*.

A CSRF attack can only be successful if the forged request happens within a previously authenticated session between the victim's browser and the

target application. Unfortunately, the design of current session management mechanisms in the browser attaches session information to any outgoing request, fostering the prevalence of CSRF attacks. For example, the browser attaches the relevant cookies for the domain, scheme and path to each outgoing request, both for requests internal to the application and cross-site or cross-application requests (illustrated in Figure 2.3). Additionally, many applications prefer long-living sessions, sticking around as long as the browser remains open, regardless of whether an application is currently active in a browser tab. Essentially, this means that if a user had an authenticated session with the target application in the lifetime of the browser, it is likely that forged requests within this authenticated session can be made.

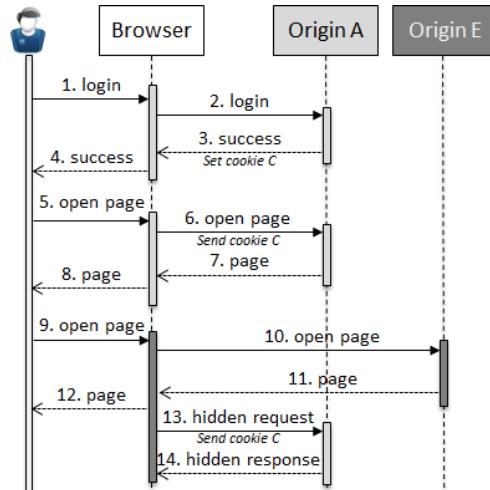


Figure 2.3: In the CSRF attack depicted here, the attacker triggers a request from origin E to the target application at origin A (step 13), to which the browser attaches the cookies from the existing session with origin A. If origin A does not have CSRF protection, this request will be executed as if it was generated by the user.

A *Login CSRF* [26] attack is a variation of a CSRF attack, where the attacker forges a request to authenticate the victim with an attacker-chosen account. Essentially, the attacker submits a login form from within the user's browser, using the credentials of the attacker. When the user unknowingly uses the targeted application, any entered information is associated with the attacker-chosen account, and can potentially leak to the attacker. A common example is a search engine keeping a history for authenticated users.

Next to traditional login CSRF attacks that forge a submission of the

target application’s authentication form, login CSRF attacks can also target applications using third-party authentication providers such as Google single sign-on, Facebook authentication or OpenID [22]. These authentication providers provide the target application with an *assertion*, containing the necessary info to confirm a successful authentication, as well as the user’s identity. Using a login CSRF attack, an attacker can submit his own assertion to the target application from the victim’s browser, effectively establishing an authenticated session tied to the attacker’s credentials.

In essence, the problem of a CSRF attack is the lack of intent, leaving the server in the dark as to whether a request was made intentionally by the end user or legitimate application code, or was forged by an attacker. The fact that browsers handle same-origin and cross-origin requests identically, and Web applications now heavily depend on this behavior, enables CSRF attacks and hampers effective countermeasures.

## Mitigation Techniques

During the early years of CSRF, several simple mitigation techniques have been proposed, but proven ineffective at protecting against CSRF attacks. One suggestion is to only carry out state-changing operations using POST requests, as actually mandated by the HTTP specification [99], assuming that forging POST requests is not feasible. Unfortunately, this is not the case [264], as shown by the code example in Listing 2.2, rendering this advice useless in protecting against CSRF.

A second mitigation technique enforces referrer checking at the server side. State-changing requests should only be accepted by the receiving application if the value of the `Referer` header,<sup>2</sup> which denotes the sending origin, contains a trusted site, and rejected otherwise. Referrer checking would effectively mitigate CSRF attacks, but unfortunately, the presence of the `Referer` header in the request headers is unreliable. The `Referer` header is often missing due to privacy concerns, since it tells the target application which resource at which URI triggered the request. Similarly, browsers do not add the header when an HTTPS resource, which is considered sensitive, refers to an HTTP resource. Additionally, browser settings, corporate proxies, privacy proxies or add-ons [1, 2] and referrer-anonymizing services [186] enable the stripping of automatically added `Referer` headers.

As an improvement to the `Referer` header, the `Origin` header provides the server with information about the origin of a request, without the strong privacy-

---

<sup>2</sup>The `Referer` header was originally misspelled in the specification, and the header has kept this name until this day. In text, the correctly-spelled `referrer` is more commonly used.

invasive nature of the `Referer` header [26]. Unfortunately, the specification [24] only states that the `Origin` header *may* be added, but does not require user agents to do so, potentially causing the same problems as with the `Referer` header. The `Origin` header, however, is mandatory when using Cross-Origin Resource Sharing (CORS) [248], an API that enables the sharing of resources across origins.

An alternative, more effective countermeasure are token-based approaches [47], for example in a hidden field of a form. A token-based approach adds a unique token to the code triggering state-changing operations. When the browser submits the request leading to the action, the token is included automatically, and verified by the server. Token-based approaches prevent the attacker from including a valid token in his payload, causing the request to be rejected. Key to the success of this mitigation technique is keeping the token for an action out of the attacker's reach. This is achieved by embedding the tokens in the application's page, where they are protected by the Same-Origin Policy, preventing theft by an attacker-controlled context, loaded in the same browser. Essential to a token-based approach is the security of the token. A token should be a hard-to-guess value, and should only be valid for a specific user. This typically requires storing the token in the user's session object at the server side, which may have a performance impact on the server.

Further research on token-based approaches, which often struggle with Web 2.0 applications, has yielded several improvements over traditional tokens, to enable complex client-side scripting and cross-origin requests between cooperating sites. jCSRF [192], a server-side proxy solution, transparently adds security tokens to client-side resources and verifies the validity of incoming requests. Alternatively, double-submit techniques [159] embed a nonce in two different locations, for example in a cookie and as a hidden form field, allowing the server to compare both values, without keeping track of state. Since the attacker cannot manipulate both tokens, he is unable to forge valid requests.

Another approach at the level of the application's architecture is based on the observation that the cross-origin accessibility of Web application resources allows the attacker to target any resource by making a request from a different origin. Several techniques propose to mitigate CSRF by restricting the set of entry points to CSRF-protected resources, thus eliminating a CSRF attack on a sensitive resource. These entry points can be enforced purely at the server side [52], or in combination with a browser-based mechanism [58]. Recently, the concept of entry points gained traction with browser vendors, and is being integrated as a core feature [206].

Another effective mitigation of CSRF attacks involves explicit user approval of state-changing operations. By requiring additional, unforgeable user interaction,

the attacker is unable to complete the CSRF attack in the background. Examples are explicit re-authentication for sensitive operations, or the use of an out-of-band device to generate security tokens, as employed by many European online banking systems. The risk associated with this mitigation technique is a shift in attack from CSRF to clickjacking, which is covered in Section 2.3.4.

Finally, client-side solutions have emerged to protect legacy applications, which are no longer updated, or where developers do not know or care about CSRF vulnerabilities, leaving users vulnerable in the end. These client-side solutions detect potentially dangerous requests and either block them, or strip them from implicit authentication credentials, such as cookies. Examples are RequestRodeo [143], the first client-side mitigation technique in the form of a proxy, followed by browser add-ons CsFire [64, 65] (covered in Chapter 3), RequestPolicy [212], DeRef [104] and NoScript ABE [168]. While these client-side solutions successfully prevent CSRF attacks, their main challenge is keeping the delicate balance between security and usability, which comes from the need to be compatible with all possible sites a user visits.

## State-of-Practice

Current practices for mitigating CSRF attacks are focused on token-based approaches, either custom-built for the application, or deployed as part of a Web framework, such as Ruby on Rails, CodeIgniter and several others. Alternatively, server-side libraries or APIs offer CSRF protection as well, such as the community-supported OWASP ESAPI API and CSRGGuard. Sites being built using a content management system (CMS) – instead of being built from scratch – can benefit from built-in CSRF support as well. For example Drupal, Django and WordPress offer token-based CSRF protection, with Drupal even extending its support to optional customized modules.

Applications using the OAuth protocol for authentication are vulnerable to login CSRF attacks, as shown by a formal analysis of Web site authentication flows [22], which has dubbed this problem as *social login CSRF*. OAuth is a protocol enabling third-party clients limited access to an API, such as used in Facebook authentication [95]. The OAuth specification recommends using a generated nonce, strongly bound to the user’s session, which would prevent a social login CSRF attack, if followed by the implementations of the protocol.

### 2.3.4 UI Redressing

A UI redressing attack, also known as clickjacking or tapjacking, redresses or “redecorates” a target application, confusing the user who is interacting with the application. For example, Figure 2.4 illustrates a clickjacking attack using a transparent overlay. UI redressing attacks can be used to trigger any user interaction within the target application, such as clicking a button, dragging-and-dropping items, etc.

A UI redressing attack uses various innocent features, combining them to trick the user into clicking a sensitive element. UI redressing attacks can be annoying, but also malicious. Examples of the former are Tweetbombs [164], which post Twitter status updates to the victim’s account, and LikeJacking, which triggers unintended likes on Facebook pages. Examples of the latter are attacks that trick the user into enabling Web cam access for the Flash Player [126], and attacks on wireless routers, stealing the secret WPA keys [210].

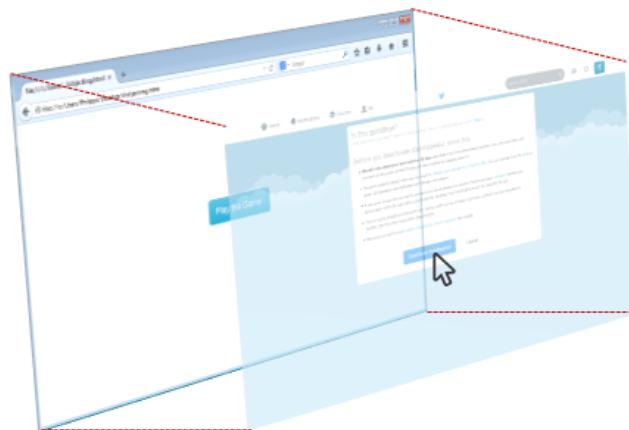


Figure 2.4: The essence of a clickjacking attack is tricking the user into clicking on a specific location, under which an element of the target application is positioned. In this example, the user thinks he starts a game, but in fact clicks on a button in the hidden target application.

#### Description

The goal of a UI redressing attack is to forge a request from the victim’s browser to the target application, by making the user unintentionally interact with an element on a page of the target application. An attacker achieves this using

misdirection, by redressing the UI of the page to hide the real element that will be clicked by the user. Many forms of UI redressing are possible, from transparent overlays to very precise positioning of elements, or even fake cursors stealing the user’s attention.

A UI redressing attack requires a coordinating application, which is under control of the attacker and actually attracts legitimate interaction from the user. However, the coordinating application masquerades the target application, causing the user’s interaction to be directed towards the target application. In the example in Figure 2.4, the user actually thinks he clicks on the *Play!* button, but in reality, the click goes towards the invisibly framed page. Since both buttons are precisely positioned on top of each other, the attacker ensures that the user actually clicks in the right location.

Note that the attacker requires the capabilities of a *Web attacker*, but does not control the target application, and that the interactions of the user with the target application are indistinguishable from legitimate interactions. Mitigation techniques for CSRF attacks are ineffective against UI redressing attacks, since the requests are not cross-origin, but actually originate from within the application.

While UI redressing attacks were traditionally known as clickjacking attacks, numerous variations have emerged as the technology evolved. *Double clickjacking* [125] tricks the user into double clicking, and quickly raises a previously opened pop-under window after the first click, misdirecting the user’s click to the target application, which is opened in the pop-under window. Another variation uses *history navigation* to preload a page of the target application in the context’s history, and subsequently navigates to the attacker page. When the user is tricked into double clicking somewhere, the attacker page quickly navigates the context to the previous page in the history, causing the user to click on a specific location in the preloaded target page. [262]. Instead of abusing clicking, an attack can also use the features of the drag-and-drop API [35] to persuade the user to drag some text into the target application, thereby injecting data into form fields [231]. Another variation is a *cursorjacking* attack [126], where a fake cursor is drawn on the screen, while the original cursor is hidden or out of the user’s focus on screen (Figure 2.5). Alternatively, *strokejacking* abuses keyboard focus features to trick the user into typing in an input field of the target application [126]. Finally, *tapjacking* brings UI redressing attacks to mobile devices, tricking the user into tapping on hidden elements [210].

In essence, a UI redressing attack results in an unintentional request by misdirecting the interaction of a user. Well-conducted UI redressing attacks are impossible to observe, exonerating the users from all blame. UI redressing

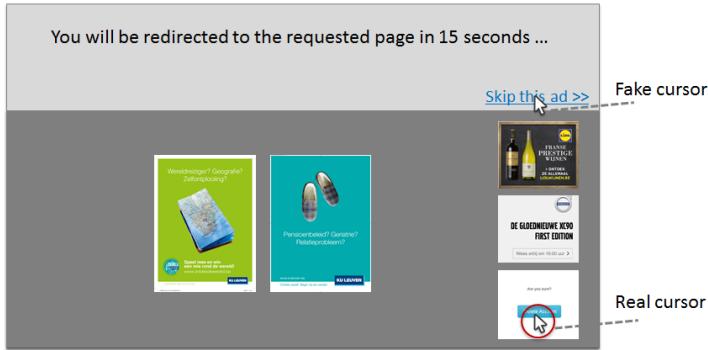


Figure 2.5: A cursorjacking attack, where the target application’s *Delete Account* button is at the bottom right of the page, with a *skip this ad* bait link remotely above it. Note there are two cursors displayed on the page: a fake cursor is drawn over the “skip this ad” link while the actual pointer hovers over the “Delete Account” button.

attacks can even be used to bypass an application’s mitigation techniques, such as an explicit confirmation request before performing sensitive actions.

## Mitigation Techniques

UI redressing attacks commonly use frames to embed the target application. Traditional mitigation techniques therefore use *framebusting code*. Framebusting code is targeted at detecting the unpermitted framing of an application, and subsequently breaking out of the frame by moving the application to a top-level frame, as shown by the example in Listing 2.3). Simple framebusting code is often easily evaded, but carefully constructing robust framebusting code can withstand evasion, or fail in safe ways [209]. The downside of framebusting is the strict on or off mode, either allowing all kinds of framing or no framing at all, not even by trusted applications. In the modern Web, with mashups and composed applications, this might be problematic.

---

```

1 //UNSAFE - DO NOT USE
2 if(top != self) top.location.replace(location);

```

---

Listing 2.3: A simple approach aimed at detecting unpermitted framing, and breaking out of the frame by moving the application to the top-level frame. While this countermeasure is often used, it is easily evaded [209].

A second popular mitigation technique is the regulation of framing through the `X-Frame-Options` header [207]. By adding this header to the response, an application can indicate that framing is denied, allowed within its origin, or allowed by an explicitly listed origin. Recently, the functionality offered by the `X-Frame-Options` header has been integrated in the Content Security Policy with the *frame-ancestors* directive [28]. The *frame-ancestors* directive offers better support for nested browsing contexts, which are still vulnerable with the `X-Frame-Options` header [157], and supports multiple host source values, instead of the one supported by the `X-Frame-Options` header.

Research on UI redressing attacks has also focused on a browser-supported solution that addresses the root cause, i.e., user misdirection. *InContext* [126] incorporates several measures that ensure that a user's click is genuine, for example by comparing screenshots at the time of the click, or by highlighting the area of the cursor to prevent attacks involving a fake cursor. *InContext* also serves as an inspiration for the new standardization efforts by W3C to ensure UI integrity in Web applications, effectively preventing UI redressing attacks [169]. The specification proposes several new directives to include in the Content Security Policy, giving the developer control over several heuristics to determine the genuineness of the interaction. Similar to other directives in the Content Security Policy, the browser will enforce these heuristics, blocking and reporting any violations.

Finally, clickjacking can be combated from the client side as well. The popular security add-on NoScript [167] includes the ClearClick module, which does a screenshot-based comparison of the area to be clicked with the actually clicked element, and which served as an inspiration for the *InContext* work [126]. When a difference between both is detected, the user is warned and explicitly asked to confirm the action, before the request is sent.

## State-of-Practice

By design, all Web applications are vulnerable to clickjacking attacks, but the attack receives little attention compared to higher-risk attacks. Users of major, well-known Web applications such as Twitter, Facebook, etc. have fallen victim to clickjacking attacks, often indicated by spam messages making their way through the application.

Many Web applications deploy some form of framebusting code, of which several variations are known to be vulnerable to evasion [209]. Additionally, many applications have a different front-end for normal browsers and mobile browsers, often only implementing framebusting in their normal version [210]. Currently, the `X-Frame-Options` header is gaining adoption. Our July 2014

survey of the Alexa top 10,000 domains discovered 2,159 domains that include an `X-Frame-Options` header in their responses.

### 2.3.5 Session Hijacking

A session hijacking attack allows the attacker to transfer an authenticated session from the victim's browser to an attacker-controlled browser. Using the transferred session, the attacker can impersonate the user, and perform all actions available to the user.

Session hijacking, together with other session-related problems, is ranked second in the OWASP Top 10 [257]. Additionally, with the ubiquitous, freely accessible, unprotected wireless networks, session hijacking has become a straightforward attack [105].

#### Description

The goal of a session hijacking attack is to transfer the user's authenticated session to a different machine or browser, enabling the attacker to continue working in the victim's session. To achieve this, the attacker hijacks the session that the user has established with the target application. Note that if the attacker manages to hijack an unauthenticated session, he simply has to wait until the user authenticates himself, since this state will be stored in the server-side session object.

Technically, once a session between the user's browser and the server is established, future requests will be handled within the context of this session. The de facto standard session management mechanism in modern Web applications is cookie-based, where a random, unique session identifier is stored in a cookie within the browser. In a session hijacking attack, an attacker succeeds in stealing the session identifier, which he can subsequently use to send requests to the server (illustrated in Figure 2.6). This is possible because the session identifier acts as a *bearer token*, and the mere presence of this identifier in a cookie attached to the request suffices for legitimizing the request within the session.

Depending on the security parameters of the cookie, an attacker has several ways of obtaining the session identifier. One way is by calling the `document.cookie` property from within the target's application origin, which can for example be achieved through cross-site scripting. Another way is by directly accessing the cookie store from a compromised browser, for example by installing a malicious

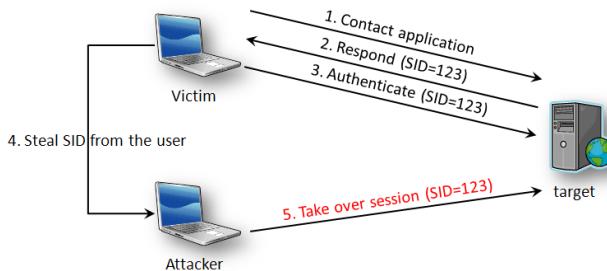


Figure 2.6: In a *session hijacking* attack, an attacker steals the session identifier of the user (step 4), resulting in a complete compromise of the user’s session.

browser add-on. A third alternative is by eavesdropping on the network traffic, and snatching the session identifier from the response, or any subsequent request, as illustrated by point-and-click tools such as Firesheep [48]. Finally, a weak or predictable session identifier can be guessed or obtained through a brute force attack.

An alternative session management mechanism is based on URI parameters, including a session identifier as a parameter in the URI in every request to the application. This mechanism is often used as a fallback mechanism for browsers that do not support cookies, or refuse to store them. Technically, little changes for a session hijacking attack, other than the means to obtain the session identifier. An attacker can still access it from JavaScript or eavesdrop on the network to extract it. Additionally, an attacker can attempt to trigger a request to an attacker-controlled resource, hoping that a `Referer` header will be included, since it contains the full URI, including the parameter with the session identifier.

In essence, a session hijacking attack is possible because the session identifier, which acts as a bearer token for an authenticated session, is easily obtained and transferable between browsers. Making the session identifier accessible through JavaScript or by eavesdropping on the network is a suboptimal decision, which enables a highly dangerous and harmful attack.

## Mitigation Techniques

A traditional mitigation technique for session hijacking is *IP address binding*, where the server binds a user’s session to a specific IP address. Subsequent requests within this session need to come from the same IP address, and any requests coming from another IP address will be discarded. While

this mitigation technique works well in scenarios where every machine has a unique, unchanging public IP address, it is ineffective when the same public IP address is shared among multiple machines, or when the public IP address changes during a session. Precisely these two cases have become ubiquitous in modern network infrastructure, with NATed home and company networks, publicly accessible, shared wireless networks and mobile networks. Recently, the technique of tracking a client has been refined through browser fingerprinting, where numerous characteristics of the browser are compiled into a fingerprint [86, 183, 4]. Anomaly detection based on the browser fingerprint triggers alerts when an unexpected fingerprint is seen, which may be an attacker stealing a session.

Another approach focuses on preventing the theft of the session identifier, which is commonly stored and transmitted in a cookie. The *HttpOnly* and *Secure* cookie attributes can be used to respectively prevent a cookie from being accessible through JavaScript, and prevent a cookie issued over HTTPS from being used (and thus leaked) on a non-HTTPS connection. Correctly applying both attributes to cookies holding a session identifier effectively thwarts script-based session hijacking attacks, as well as session hijacking attacks through eavesdropping on network traffic.

One long-lived line of research focuses on providing protection against session hijacking attacks from within a Web application, not by hiding the session identifier, but by ensuring that the session identifier no longer acts as a bearer token, meaning that the mere knowledge of the session identifier is insufficient to hijack a session.

*SessionSafe* [140] combines several mitigation techniques against session hijacking into a single countermeasure, and thereby effectively prevents script-based session hijacking attacks. To summarize, the three combined mitigation techniques are (i) deferred loading, which hides the session identifier from malicious JavaScript before main content is loaded, (ii) one-time URLs, where a secret component prevents URLs from being guessed by an attacking script, and (iii) subdomain switching, which removes the implicit trust between pages that happen to belong to the same origin, but not necessarily trust one another.

*SessionLock* [5] negotiates a shared secret between client and server, and stores this in the client-side context. The secret is used to add integrity checks to outgoing requests. Since the secret value is never transmitted in the clear, SessionLock prevents an attacker with a stolen session identifier from making valid requests. Unfortunately, because the secret is stored in the JavaScript context, it cannot be protected against script-based attacks.

The *HTTP Integrity Header* [112] uses a similar approach as SessionLock, but

makes the secret negotiation and integrity check part of the HTTP protocol, thereby avoiding modifications to the application logic. *SecSess* [71] further improves on the HTTP Integrity header by achieving compatibility with commonly deployed middleboxes, such as Web caches (covered in Chapter 5). *GlassTube* [113] also ensures integrity on the data transfer between client and server, and can be deployed both within an application or as a modification of the client-side environment, for example as a browser plugin.

Finally, other approaches look into strengthening cookies to prevent session hijacking attacks. *One-Time Cookies* [59] propose to replace the static session identifier with disposable tokens per request, similar to the concept of Kerberos service tickets. Each token can only be used once, but using an initially shared secret, every token can be separately verified and tied to an existing session. *Macaroons* [38] improve upon cookies by placing restrictions on how, where and when the implicit authority of the bearer token can be used. The technology behind macaroons is based on chains of nested HMACs, built from a shared secret and a chain of messages. Macaroons target cloud services, where delegation between principals without a central authentication service is often required, for example for sharing access to the user’s address book on another service.

Other techniques follow a similar approach, but base their security measures on the user’s password, which is in itself a shared secret between the user and the Web application. *BetterAuth* [142] revisits the entire authentication process, offering secure authentication and a secure subsequent session. *Hardened Stateless Session Cookies* [181] use unique cookie values, calculated using hashing functions based on the user’s password, effectively preventing the generation of new requests within an authenticated session.

Alternatively, *Origin-Bound Certificates* (OBC) [81] extend the TLS protocol to establish a strong authentication channel between browser and server, without falling prey to active network attacks. Within this secure channel, TLS-OBC supports the binding of cookies and third-party authentication tokens, which prevents the stealing of such bearer tokens.

A final line of research targets session hijacking problems from the client side, without explicit support from the target application. *SessionShield* [185] is a client-side proxy that mitigates script-based session hijacking attacks by ensuring all session cookies are marked *HttpOnly* before they reach the browser. Determining which cookies are session cookies at the client side, in an application-agnostic way, is achieved by applying sensible heuristics, including an entropy test.

## State-of-Practice

Unfortunately, many sites still use unprotected cookies to store session identifiers, leaving users vulnerable to session hijacking attacks. On the bright side, the adoption of the *HttpOnly* and *Secure* attributes is gaining ground, starting to be turned on by default [17]. Our July 2014 survey of the Alexa top 10,000 domains shows that more than half of the domains use the *HttpOnly* attribute (5,465 domains in total), and 1,419 domains use the *Secure* attribute, which is a significant increase compared to a study from 2010 [223].

Another practice that is being deployed by major sites is to operate split session management between HTTP- and HTTPS-accessible parts of the site. For example, a Web shop can run its catalog inspection and shopping cart filling operations over HTTP, and use HTTPS for sensitive operations, such as logging in, checking out the cart, payments or account modifications. Technically, they use two different session cookies, one for HTTP usage and one for HTTPS usage, where the latter is declared *HttpOnly* and *Secure*. While this leaves the user vulnerable to a session hijacking attack on the HTTP part, it effectively protects the HTTPS part, where sensitive operations are conducted.

### 2.3.6 Session Fixation

A session fixation attack enables an attacker to force the victim's browser to use an existing session, which is also known by the attacker. The goal of the attacker is to wait for the user to perform state-changing actions, such as authenticating himself to the application, after which the attacker takes control of the session. The effects of a session fixation attack are similar to those of a session hijacking attack.

Session fixation is categorized as a session management problem, ranked second in the OWASP Top 10 [257]. Session fixation is technically more difficult than session hijacking, and requires the capability to transfer a session identifier towards the victim's browser. Unfortunately, no exact numbers of the prevalence of session fixation attacks are available. However, the prevalence of the attack vectors that can lead to a session fixation attack are a good indicator, which is covered during the discussion of related attacks.

## Description

The goal of a session fixation attack is to register the results of the victim's state-changing actions in a session controlled by the attacker. The most prominent example of such a state-changing action is the authentication process, which results in the authentication state being stored in the session. The attacker forces the victim's browser to use a specific, attacker-known session, allowing him to retake control of this session at any time. If the attacker takes over the session after user authentication, he can effectively impersonate the user.

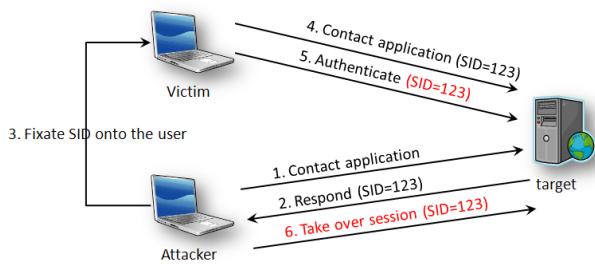


Figure 2.7: In a *session fixation* attack, an attacker fixates his own session identifier into the browser of the user (step 4), causing the user to authenticate in the attacker's session.

For cookie-based session management systems, the attacker first obtains a valid session identifier for the application, either by visiting the target application himself, or by crafting a session identifier. In the next step, the attacker has to fixate the session identifier in the victim's browser, which depends on the session management mechanism used by the application. Once the session is fixated and the user visits the application, the user will be working within the attacker's session. This means that the authentication state at the server side will be stored within this session as well, allowing the attacker to take over the session later on (illustrated in Figure 2.7).

The crucial part of a session fixation attack is fixating the session identifier, an action that depends on the presence of a secondary vulnerability, such as cross-site scripting, header injection, etc. [74]. For example, in cookie-based session management systems, the attacker can set a cookie using the `document.cookie` property from JavaScript, or by using an injection attack to insert `meta` elements that mimic header operations into the page's content, or by manipulating network traffic.

Fixating a session identifier in parameter-based session management systems is straightforward. All it takes is tricking the user into visiting a URI which

contains the fixated session identifier as a parameter in the query string.

In essence, session fixation attacks are possible because the session identifier acts as a bearer token for a session between a user and an application, combined with the fact that sessions are easily transferable between browsers. Session fixation and session hijacking attacks both exist for the same reasons, but use a different attack vector to obtain an authenticated session.

## Mitigation Techniques

Due to the multiple attack vectors that can lead to a session fixation attack, plugging them all is difficult. Nonetheless, protecting session cookies with the *Secure* and *HttpOnly* attributes makes the attack more difficult, since it prevents an attacker from easily overwriting an already existing session cookie. However, these protections can be bypassed, for example by overflowing the cookie jar with meaningless cookies [13], causing the browser to purge the oldest ones (i.e., the session cookie), allowing the attacker to fixate a new session identifier.

An effective mitigation technique for fixation attacks consists of sending the user a renewed session identifier after the user changes privilege levels in the application, such as a login or logout operation, accessing an administrative part of the application, etc. For example, by issuing a new session identifier after user authentication, an application ensures that the authentication information is not associated with the fixated session identifier, preventing the attacker from taking over the authenticated session. Renewing the session identifier is the server's responsibility, and is often supported by the Web programming language or Web framework. No explicit client support is required, since the server can just override the already-existing session cookie using a *Set-Cookie* header.

However, integrating the renewal of the session identifier in legacy applications is challenging. Researchers have proposed several solutions to this problem, both from the server side and the client side. A first solution is to integrate the renewal of the session identifier directly in the server-side framework, as part of the session management mechanism [141]. Alternatively, the same approach can be offered as a server-side reverse proxy, which scans requests and responses, and appends an additional, protected session identifier when required [141]. On the other hand, our client-side protection mechanism against session fixation attacks, called *Serene*, offers protection to a user, without requiring a change or modification at the server side [74] (covered in Chapter 4). *Serene* is a browser add-on that detects cookie and parameter-based session identifiers in requests and responses, and offers additional protection for these identifiers. *Serene* can

prevent cookie-based session fixation attacks through an injection attack vector, as well as parameter-based session fixation attacks.

A variant of a session fixation attack can be carried out by a *related domain attacker* [41], who controls an application hosted under the same registered domain as the target application (e.g., *example.com*). By setting a cookie that applies to all sibling domains, an attacker can easily fixate a session identifier. *Origin Cookies* [41] protect applications against this kind of attack by allowing cookies to be limited to one domain, preventing manipulation from sibling sites. *Origin-Bound Certificates* [81] is follow-up research that offers even stronger guarantees, but requires a TLS-secured channel to be present.

### State-of-Practice

Modern frameworks all support the renewal of the session identifier, albeit only after explicit actions from the developer to enable this behavior [222]. Additionally, correctly enabling the *HttpOnly* attribute on session cookies can successfully mitigate certain attack vectors.

### 2.3.7 Cross-Site Scripting

With a Cross-Site Scripting (XSS) attack, an attacker is able to execute his own JavaScript code within the application's execution context, gaining him the same privileges as the target application code. This exposes all client-side application data, resources and APIs to the attacker, including the possibility to manipulate and generate legitimate application requests towards the server-side application code.

Cross-site scripting is a serious problem in the Web, and is highly ranked in both the OWASP top ten of Web application vulnerabilities [257] and the CWE/SANS Most dangerous programming errors [172]. Almost every Web application on the Web has had a script injection vulnerability at some point, with even the serious players such as Google, Facebook and Twitter not being exempted [260]. Hence, cross-site scripting is often referred to as the *buffer overflow of the Web*.

## Description

The goal of a cross-site scripting (XSS) attack is to execute attacker-controlled code in the client-side application context within the victim's browser. In an XSS attack, the attacker is able to inject JavaScript code into a page of the target application, mixing it with the legitimate page content, causing it to be executed altogether as the page is processed. Since the browser sees a single Web page, it is unable to distinguish between legitimate code and malicious code. XSS attacks can be carried out by the weakest threat model, the *forum poster*.

An attacker has many attack vectors to inject a payload into the target application. A first way is by manipulating the URI to inject code into request parameters, which are processed by a client-side script of the target application. Whenever the client-side script constructs code using these parameters, which it does not expect to hold code, the attacker's code will be executed alongside the legitimate application code. This type of XSS attack is known as *DOM-based XSS* or *XSS type 0*.

A second class of XSS attacks consists of tricking the server into including the attacker's code in its response. For example, if the attacker makes the victim's browser visit the URI shown in Figure 2.8, the server will reflect the value of the URI parameter back in the response, where it will be executed as part of the requested page. This type is known as *reflected XSS* or *XSS type 1*.

Finally, an attacker can also store the malicious code in the application's data, for example by hiding it in a forum post or blog comment. Whenever the victim requests a page that includes the attacker's content, the malicious code will be embedded in the page as well. This type of XSS is known as *stored XSS* or *XSS type 2*, and is illustrated in Figure 2.9.

```
http://example.com/search.php?q=%3Cscript%3Ealert(%22XSSed!%22)%3C%2Fscript%3E
```

Figure 2.8: When the vulnerable Web application processes this URI, the source of the response will include `<script>alert("XSSed!")</script>`, leading to a reflected cross-site scripting attack.

In essence, the problem of an XSS attack is the failure of the target application to recognize the insertion of code, thus allowing the payload to be executed. The combination of the facts that code can be placed anywhere in a document, and that browsers attempt to correct syntactically incorrect documents rather than rejecting or ignoring them helps the easy exploitation of injection vulnerabilities.

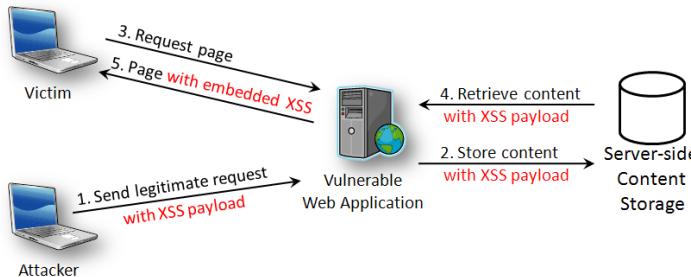


Figure 2.9: In a stored cross-site scripting attack, the attacker injects script code into the application’s server-side content storage, which is then unknowingly served to victim users, visiting legitimate pages of the application.

## Mitigation Techniques

The traditional mitigation technique used against cross-site scripting attacks depends on sanitizing input and output, preventing any dangerous input from reaching the final output. These sanitization techniques attempted to simply replace or remove dangerous characters such as < > & " ' or check against a whitelist of allowed characters, but modern sanitization techniques take the context of the output into account.

Modern Web applications generate output for different contexts, with different output formats and injection vectors. Some example contexts are HTML elements, HTML element attributes, CSS code, JavaScript code, etc. Several publicly available libraries provide context-sensitive content encoding, and effectively mitigate XSS attacks. Popular examples for Java applications are the OWASP Java Encoder Project [131], which offers several context-specific sanitization operations, and OWASP’s Java XML Templates [130], which offers automatic context-aware encoding. Alternatively, HTML Purifier [261] offers automatic sanitization for PHP applications, and even ensures that the output is standards-compliant HTML. Automating context-sensitive sanitization is an active research topic. ScriptGard [216] focuses on the detection of incorrect use of sanitization libraries (e.g., context-mismatched sanitization or inconsistent multiple sanitizations), and is capable of detecting and repairing incorrect placement of sanitizers. Other work focuses on achieving correct, context-sensitive sanitization, using a type-qualifier mechanism to be applied on existing Web templating frameworks [213].

Even with the most advanced mitigation techniques, both newly created and legacy applications remain vulnerable to XSS attacks. Therefore, Mozilla

proposed Content Security Policy (CSP) [227], a server-driven, browser-enforced policy to be used as a second line of defense. CSP allows a developer or administrator to strictly define the sources of included content, such as scripts, stylesheets, images, etc., preventing the inclusion of malicious scripts from sources unknown to or untrusted by the developer. Additionally, CSP prevents the execution of harmful inline content by default. When deploying CSP, a reporting-only mode is available. This mode will report any violations of the policy to the developer, without actually blocking any content. This allows to dry-run a policy before actually deploying it towards users.

CSP's restrictions on dangerous inline content effectively render injected script code harmless, since it will not be executed, and the list of trusted sources further limits an attacker when including a remote script file. Currently, CSP is being adopted by the major browsers, and is on the standardization track of W3C [229]. One downside of CSP is its impact on an application's code, since the application is no longer allowed to use code files that mix HTML and JavaScript together, or dangerous features that interpret strings into code at runtime, such as the *eval()* function. For newly developed applications, this is manageable, but legacy applications require some effort to be made compatible [254]. As a response to this problem, the upcoming level 2 version of CSP [28] will allow inline scripts if they possess a unique, unguessable nonce. Injected scripts will not be able to provide this nonce, and hence will not be executed.

The detection of cross-site scripting vulnerabilities in Web applications commonly relies on penetration testing (colloquially referred to as *pentesting*) and static analysis [124, 98]. In addition to these state-of-practice techniques, state-of-the-art research focuses on the discovery and detection of potential injection vulnerabilities. Kudzu [215] achieves this using symbolic execution of JavaScript. Gatekeeper [109], on the other hand, allows site administrators to express and enforce security and reliability policies for JavaScript programs, and was successfully applied to automatically analyze JavaScript widgets, with very few false positives and no false negatives.

## State-of-Practice

Injection vulnerabilities leading to XSS attacks are prevalent in both new and legacy Web applications. A large scale analysis of the Alexa top 5,000 sites has discovered 6,167 unique XSS vulnerabilities, distributed over 480 domains [158]. Cross-site scripting attacks are often only the first step in a more complicated attack, involving underlying infrastructure or higher-privilege accounts. The consequences of escalating an XSS attack are aptly demonstrated by exploitation

frameworks, such as the *Browser Exploitation Framework* (BeEF) [12] or *Metasploit* [200].

Currently, almost every newly developed Web application sanitizes its inputs and outputs, in an attempt to avoid injection vulnerabilities altogether. Most modern development frameworks offer library support for sanitization. Unfortunately, sanitization libraries are not always context-sensitive, and many applications apply sanitization procedures wrongly or inconsistently [216]. Additionally, a few context-sensitive sanitization libraries are available, as discussed above as an aspect of mitigation techniques.

Since injection vulnerabilities remain widespread, several attempts have been made to stop them from within the browser, independent of any application-specific mitigation techniques. Examples of in-browser mitigation techniques are XSS filters [29, 205, 230], or the popular security add-on NoScript [167]. The newly introduced Content Security Policy (CSP) [229, 28] is slowly starting to be adopted. Our July 2014 survey of the Alexa top 10,000 sites found 131 sites that already issue a CSP policy in their response headers.

Additionally, applications often apply code-based isolation techniques to prevent the damage that can be done by untrusted or injected scripts. Examples of currently available isolation techniques are HTML 5 sandboxes [36], or browser-based sanitization procedures for dynamic script code, such as Internet Explorer’s `toStaticHTML()` [135].

### 2.3.8 Social Engineering Attacks

Social engineering focuses on people as the attack vector, using psychological manipulation techniques to trick people into performing certain actions or into divulging confidential information. Within the Web context, social engineering attacks are generally aimed at gaining control over the user’s account, financial information or identity information.

Social engineering is part of human nature, and has existed long before the Web. However, due to the Web’s distributed nature, social engineering attacks have become easier and more widespread. One of the most common examples is *phishing*, where unsuspecting users are tricked into entering sensitive information, such as their credentials or credit card information, into a fraudulent authentication form. PhishTank, an anti-phishing initiative, collects about 20,000 valid phishing Web sites per month [190]. Another target of social engineering attacks are companies that manage the users’ accounts, resulting in compromised Twitter accounts [119], wiped Apple devices [123], etc.

## Description

The goal of a social engineering attack is to gain access to confidential information, which allows the attacker to escalate the attack, for example by taking control over the user's online accounts, by charging fraudulent transactions to the user's credit cards, or by committing identity theft by impersonating the user. In the context of this dissertation, the focus lies on the theft of credentials, subsequently used to authenticate to a service in the user's name. Especially in the modern, interconnected Web, compromise of one account often allows the escalation towards other accounts, and the victim's entire online presence [123, 119].

Attackers employ social engineering techniques to trick victims into willingly surrendering their credentials. For example in a *phishing* attack, the attacker capitalizes on a user's inability of distinguishing a legitimate page from one that looks legitimate but is actually fraudulent. By luring the user to the fraudulent page, for example with a carefully crafted "urgent" email message, the user is tricked into entering his credentials, causing them to be sent to the attacker. Phishing attacks can be conducted both on large and small scale, depending on an attacker's objectives. Large scale attacks are very generic, and generally easier to detect. Small scale attacks, also known as *spear phishing*, target highly specific individuals and companies, and are very difficult to detect.

A variation on the traditional phishing attack is *tabnabbing* [201]. In tabnabbing (shown in Figure 2.10), the user is lured into visiting a malicious site, which however looks innocuous. If a user keeps the attacker's site open and uses another tab of his browser to browse to a different Web site, the tabnabbing page takes advantage of the user's lack of focus (accessible through JavaScript as `window.onBlur`) to change its appearance (page title, favicon and page content) to look identical to the login screen of a popular site. When a user returns back to the open tab, he has no reason to re-inspect the URL of the site rendered in it, since he already did that in the past. This type of phishing separates the opening of a site from the actual phishing attack and could, in theory, even trick users who would not fall victim to traditional phishing attacks.

In essence, social engineering attacks take advantage of the user's inability to spot a fraudulent Web application, still a challenging task in the modern Web. Additionally, a user's credentials are very valuable to the attacker, and traditional username/password-based credentials are easily transferable, often reused and stored insecurely.

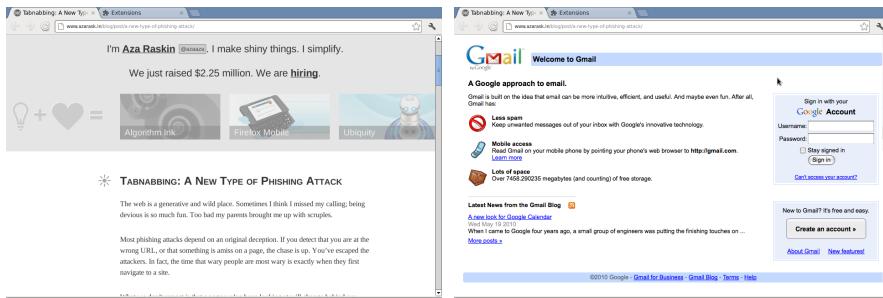


Figure 2.10: In a *tabnabbing* attack, the attacker switches an innocuous-looking tab (left) to a phishing page (right), avoiding being caught when the user checks the URI of a newly loaded tab.

## Mitigation Techniques

A recent evolution towards limiting the impact of credential theft through social engineering is the use of multi-factor authentication. In a multi-factor authentication process, the application no longer depends on a single piece of knowledge, such as a set of credentials, but requires additional factors, such as a token sent to a user's phone by text message, a token generated by a dedicated device [90], a smart card, biometric information, etc. Multi-factor authentication makes the traditional credentials less valuable, since one of the additional authentication factors is an out-of-band device, beyond the control of an attacker. However, introducing additional authentication factors also introduces additional concerns. For example, if the user's smartphone acts as a second factor in the authentication process, a problem arises when the phone is stolen, since it provides both the browser, with potentially stored credentials, and the out-of-band device. Similarly, biometrics are often considered a viable alternative to password authentication [18, 33], but they possess different characteristics compared to traditional credentials. For example, fingerprints are left behind everywhere, and the readers can easily be fooled [204]. Additionally, the amount of biometric information is limited (i.e. 10 fingerprints), and revocation is rather difficult.

In addition to multi-factor authentication, major sites further improve their authentication procedures with additional security checks when logging in from an untrusted device, similar to anomaly-based prevention of credit card fraud. Microsoft, Facebook and Google allow you to register trusted computers, from where a traditional username/password-based authentication can be used. All other machines will require two-factor authentication with a verification

code [96, 107].

Attackers have been trying to convince users to voluntarily give up their credentials for at least the last 19 years [152]. Several studies have been conducted, trying to identify why users fall victim to phishing attacks [78, 88] and various solutions have been suggested, such as the use of per-site “page-skinning” [77], security toolbars [259], images [7], trusted password windows [214], use of past-activity knowledge [184] and automatic analysis of the content within a page [265]. Finally, users can also install client-side countermeasures to protect themselves against phishing [55] and tabnabbing [73].

### **State-of-Practice**

In practice, stolen credentials and financial information are a valuable asset, as illustrated by the high demand on underground markets [134]. To prevent credential abuse, major Web sites offer strong, multi-factor authentication, in combination with trusted devices, which effectively mitigates most of the risk associated with credential theft. Additionally, the major players such as Google and Facebook, also offer single-sign on solutions, allowing other sites to benefit from the secure authentication procedures. On the downside, numerous smaller sites still use traditional credentials, and cannot prevent the use of stolen credentials.

Unfortunately, combating phishing in an automated way is difficult, which is why the currently deployed anti-phishing mechanisms in popular browsers are all black-list based [76]. The blacklists themselves are either generated automatically by automated crawlers, searching for phishing pages on the Web [255] or are crowdsourced [190]. Similarly, major corporations and financial institutions employ security firms that manually look for phishing and impersonation pages, allowing a quick flagging and removal process.

## **2.4 Contributions Revisited**

The state-of-practice in defending against common client-side attacks is less than stellar. Only 34% of the top 10-million sites use a valid TLS certificate [251], and the SSL Pulse project shows that many TLS deployments are suboptimal [198]. Additionally, the adoption of straightforward, low-impact countermeasures, such as the *HttpOnly* cookie attribute, is slow [67], and the exploitation of CSRF and XSS attacks is rampant, even on major, security-conscious sites [145, 149, 260]. Finally, users are frequently targeted by social engineering attacks, of which phishing remains the most common attack vector [190].

The whole Web session concept and supporting session management mechanisms, crucial building blocks of practically every Web application, are exceptionally vulnerable to these kind of attacks. Attacks such as session hijacking and session fixation, which use eavesdropping or cross-site scripting as attack vectors, directly target the session management mechanism. Other attacks, such as cross-site request forgery or UI redressing attempt to manipulate the user's session by carrying out additional actions in the user's name. Finally, social engineering attacks towards the user divulge the user's credentials to the attacker, allowing the attacker to fully impersonate the user towards a Web application by establishing an authenticated session.

In this section, these problems related to the security of Web sessions and session management are further investigated. Three concrete threats against the security of Web sessions are presented, followed by an overview of how our contributions effectively counter these threats.

### 2.4.1 Scope

Many of the attacks presented in the previous section target Web sessions and session management mechanisms in one way or another. At the roots of these attacks lie three concrete threats against Web sessions and session management mechanisms. The specificities of each threat are explained and related back to the attacks discussed earlier. The threats are presented in increasing order of severity.

#### **Violating Session Integrity**

The first threat is the violation of the *integrity of a session*, as encountered in other papers [26, 10], where an attacker is able to manipulate the session state. For example, if an attacker can remove requests from a session, or insert requests into the session, the integrity of the session is compromised. Note that the attacker is not assumed to have full control over the session, which would allow a transfer to another machine or browser, which is a significantly more powerful attack, as will become clear in the next threat.

A first attack that violates the integrity of the session is cross-site request forgery (CSRF), where an attacker tricks the user's browser to send requests to the target application, which interprets these requests as legitimate. A second attack is UI redressing, where the user is tricked into interacting with a seemingly innocuous page, while in fact he is interacting with a hidden page of the target application. In a third attack scenario, an attacker is able to take control of the

client-side execution context, for example through a cross-site scripting (XSS) attack, allowing him to send arbitrary requests to the application’s origin. Note that the main difference between a cross-site scripting attack and a cross-site request forgery (CSRF) attack is the level of control over the origin of the target application, since in a CSRF attack, the attacker mainly controls his own origin, and not that of the application under attack.

### **Unauthorized Transfer of a Session**

The second threat to the security of Web sessions is the *unauthorized transfer of a session*, essentially allowing an attacker to take control over the user’s session. If an attacker succeeds in taking over a user’s session, he can impersonate the user towards the target application, giving the attacker the same privileges as the user. While this threat is more powerful than violating the integrity of a session, the attacker is still bound within this single session, losing his access when the session is terminated, or when a re-authentication is required by the target application.

The most common way to perform an unauthorized session transfer is a session hijacking attack, by simply stealing the session identifier. Because this session identifier acts as a bearer token, the attacker can successfully impersonate the user towards the target application. A second attack session fixation, which has the same result as a session hijacking attack, but is technically more complicated to carry out. In a session fixation attack, the attacker first establishes a session with the target application, and subsequently transfers this session to the user’s browser, causing the user’s actions to be carried out within the attacker’s session. When the user authenticates himself within this session, the attacker gains access to the user’s authenticated session, giving him the same privileges as the user.

Improving session management in the Web is an active research topic, and many proposals effectively mitigate the unauthorized transfer of a session [5, 59, 112, 142], albeit without explicitly naming the security property. One paper [41] that investigates security challenges when two applications are hosted on a sibling domain defines a violation of *session confidentiality* as the attacker learning the session identifier, and *session integrity* as the attacker being able to modify the session identifier, which respectively corresponds to a session hijacking and session fixation attack. Even though these definitions are explicitly tailored towards session management mechanisms that use bearer tokens, our more generic definition of the threat inherently subsumes these bearer token-based definitions.

### Impersonating the User by Establishing a Session

A third threat to the security of a session is even more powerful, and allows an attacker to establish a new session in the user's name. In such an attack, the attacker can fully impersonate the user, bypassing all re-authentication checks that are based on the credentials used to establish the session. Note that this attack can escalate towards other applications as well, either because of shared credentials between multiple applications, or because of the use of the compromised application to gain access to another application. A common example of the latter is an attacker controlling a mail account, which is in turn used to reset the password of other accounts.

The most common example of establishing a session in the user's name is the use of stolen credentials, which can be obtained in various ways. Server-side compromises, resulting in the theft of a database with user information, are extremely common [132, 202]. Client-side examples of potential attacks are generally based on social engineering, where the user is tricked into divulging sensitive information. The most common attack vector for social engineering is phishing in all its variations, such as large-scale email campaigns or targeted spear phishing, but also alternative attacks such as tabnabbing.

Note that social engineering attacks, and especially phishing attacks, are very powerful when launched from within the right context. One recent example [145] is the use of a cross-site scripting vulnerability in the targeted application to redirect the user to a legitimate-looking but fraudulent login page. As this conforms with the expected flow of events, this attack is unlikely to be detected by a run-of-the-mill Web user.

#### 2.4.2 Solutions

Each of these threats against Web sessions targets a different aspect of the session, and is enabled by different attacks. Each of these attacks is in turn enabled by specific threat models, which depend on different technologies or design properties of the Web. Unfortunately, there is no silver-bullet approach that would fix all session problems at once, not even when the entire session management mechanism is replaced by an alternative approach. Within this dissertation, we have consistently improved the security of Web sessions and session management mechanism, either from within the browser as an autonomous client-side mitigation technique, or by proposing an alternative, secure-by-design solution.

*CsFire* is a client-side mitigation technique against CSRF attacks, which *violate*

*the session integrity.* CsFire prevents cross-origin requests, the kind of requests that are used in a CSRF attack, from being associated with an existing session, thereby preventing the action from being carried out in the user’s name.

The second threat, *unauthorized transfer of a session*, is enabled by the lax security properties of the session identifier, which acts as a bearer token in currently deployed session management mechanisms. Serene mitigates several attack vectors of a session fixation attack from within the browser. Unfortunately, bearer-token based session management systems are inherently insecure, and cannot be completely fixed by applying patches. Therefore, we propose SecSess, an alternative session management mechanism, which no longer depends on a bearer token, and effectively mitigates the *unauthorized transfer of a session* after establishment.

The third threat, *impersonating the user by establishing a session*, is enabled by the theft of credentials, for example through social engineering. TabShots is a detection mechanism for tabnabbing attacks, a sneaky variation on traditional phishing attacks. By detecting such attacks, and highlighting potentially fraudulent forms, TabShots effectively helps preventing the theft of credentials autonomously from within the browser.

In the upcoming four chapters, each of these contributions is discussed in detail.

## Chapter 3

# Protecting Users against Cross-Site Request Forgery

The paper covered in this chapter proposes *CsFire*, a client-side countermeasure against cross-site request forgery (CSRF) attacks. *CsFire* prevents cross-origin requests from violating the *session integrity*, by using an in-browser request filtering algorithm to distinguish between expected and unexpected cross-origin requests. By means of a bounded-scope model checking tool, we verified the effectiveness of our algorithm against the threat model of a CSRF attack, also known as the *Web attacker*. This paper was presented at the *16th European Symposium on Research in Computer Security* (ESORICS 2011) [65],<sup>1</sup> and builds upon earlier research, titled *Transparent Client-side Mitigation of Malicious Cross-domain Requests*, presented at the *2nd International Symposium on Engineering Secure Software and Systems* (ESSoS 2010) [64].<sup>2</sup>

Based on the research prototype from both papers, we continued development on *CsFire*, and released it as a freely available browser add-on, initially for Mozilla Firefox, and later for Google Chrome as well. *CsFire* attracted the attention of security and privacy aware Web users, causing the user base to grow organically to about 2,500 to 3,000 daily users at the time of this writing. In August 2010, *CsFire* was even featured as one of the security tools on the software DVD of the German computer magazine CHIP.

Our work on *CsFire* has also inspired further research on the integrity of Web sessions. Braun et al. [43] extend the request filtering algorithm with a

---

<sup>1</sup>Philippe De Ryck is the lead author of this paper.

<sup>2</sup>Philippe De Ryck is the lead author of this paper.

lightweight version of the Cross-Origin Resource Sharing (CORS) policy [248], resulting in a final decision whether to allow authentication data or not. Bugliesi et al. [45] propose a provably sound mechanism to protect session integrity, which is implemented in their browser add-on *SessInt*. The authors acknowledge the problem of false positives, and refer to CsFire’s policies as a potential solution for a more fine-grained approach. Finally, a recent standardization proposal introduces *Entry Point Regulation for Web Apps* [206], which is a browser-enforced policy that restricts cross-origin requests to a pre-defined list of entry points. This approach is comparable to CsFire’s configurable policies, where an application’s entry points can be marked as *Allowed*, and other requests as *Blocked*.

In hindsight, we can conclude that CsFire pushes the balance between usability and security to its limits, by deploying a sophisticated detection policy, while offering strong protection against CSRF attacks, as illustrated by the thousands of daily users. In the 4 years since CsFire’s release, we have noticed that the dynamics of the Web have become even more interconnected and tightly integrated, which CsFire can address with the server-driven policy rules. However, as this trend is likely to continue in the coming years, it may be useful to investigate a new security model, which goes beyond the commonly-used origins. By dividing the user’s sites into different zones, for example a *low-security zone* and a *sensitive information zone*, CsFire can prevent unexpected interactions between these zones, while allowing sites within the same zone to freely interact with each other. Such an approach also applies to corporate scenarios, where browsers are used to visit sites in a *public internet zone* and a *private intranet* zone, and where unexpected interactions between both are undesired.

# Automatic and Precise Client-side Protection against CSRF Attacks

**Abstract** A common client-side countermeasure against Cross Site Request Forgery (CSRF) is to strip session and authentication information from malicious requests. The difficulty however is in determining when a request is malicious. Existing client-side countermeasures are typically too strict, thus breaking many existing Websites that rely on authenticated cross-origin requests, such as sites that use third-party payment or single sign-on solutions.

The contribution of this chapter is the design, implementation and evaluation of a request filtering algorithm that automatically and precisely identifies expected cross-origin requests, based on whether they are preceded by certain indicators of collaboration between sites. We formally show through bounded-scope model checking that our algorithm protects against CSRF attacks under one specific assumption about the way in which good sites collaborate cross-origin. We provide experimental evidence that this assumption is realistic: in a data set of 4.7 million HTTP requests involving over 20,000 origins, we only found 10 origins that violate the assumption. Hence, the remaining attack surface for CSRF attacks is very small. In addition, we show that our filtering does not break typical non-malicious cross-origin collaboration scenarios such as payment and single sign-on.

## 3.1 Introduction

From a security perspective, Web browsers are a key component of today’s software infrastructure. A browser user might have a session with a trusted site A (e.g. a bank, or a Webmail provider) open in one tab, and a session with a potentially dangerous site B (e.g. a site offering cracks for games) open in another tab. Hence, the browser enforces some form of isolation between these two origins A and B through a heterogeneous collection of security controls collectively known as the *same-origin-policy* [263]. An *origin* is a (protocol, domain name, port) triple, and restrictions are imposed on the way in which code and data from different origins can interact. This includes for instance restrictions that prevent scripts from origin B to access content from origin A.

An important known vulnerability in this isolation is the fact that content from origin B can initiate requests to origin A, and that the browser will treat these requests as being part of the ongoing session with A. In particular, if the session

with A was authenticated, the injected requests will appear to A as part of this authenticated session. This enables an attack known as *Cross Site Request Forgery (CSRF)*: B can initiate effectful requests to A (e.g. a bank transaction, or manipulations of the victim’s mailbox or address book) without the user being involved.

CSRF has been recognized since several years as one of the most important Web vulnerabilities [26], and many countermeasures have been proposed. Several authors have proposed server-side countermeasures [26, 47, 144]. However, an important disadvantage of server-side countermeasures is that they require modifications of server-side programs, have a direct operational impact (e.g. on performance or maintenance), and it will take many years before a substantial fraction of the Web has been updated.

Alternatively, countermeasures can be applied on the client-side, as browser addons. The basic idea is simple: the browser can strip session and authentication information from malicious requests, or it can block such requests. The difficulty however is in determining when a request is malicious. Existing client-side countermeasures [143, 64, 165, 166, 211, 264] are typically too strict: they block or strip all cross-origin requests of a specific type (e.g. GET, POST, any). This effectively protects against CSRF attacks, but it unfortunately also breaks many existing Websites that rely on authenticated cross-origin requests. Two important examples are sites that use third-party payment (such as PayPal) or single sign-on solutions (such as OpenID). Hence, these existing client-side countermeasures require extensive help from the user, for instance by asking the user to define white-lists of trusted sites or by popping up user confirmation dialogs. This is suboptimal, as it is well-known that the average Web user cannot be expected to make accurate security decisions.

This chapter proposes a novel client-side CSRF countermeasure, that includes an automatic and precise filtering algorithm for cross-origin requests. It is *automatic* in the sense that no user interaction or configuration is required. It is *precise* in the sense that it distinguishes well between malicious and non-malicious requests. More specifically, through a systematic analysis of logs of Web traffic, we identify a characteristic of non-malicious cross-origin requests that we call the *trusted-delegation assumption*: a request from B to A can be considered non-malicious if, earlier in the session, A explicitly delegated control to B in some specific ways. Our filtering algorithm relies on this assumption: it will strip session and authentication information from cross-origin requests, unless it can determine that such explicit delegation has happened.

We validate our proposed countermeasure in several ways. First, we formalize the algorithm and the trusted-delegation assumption in Alloy, building on the formal model of the Web proposed by [10], and we show through bounded-scope

model checking that our algorithm protects against CSRF attacks under this assumption. Next, we provide experimental evidence that this assumption is realistic: through a detailed analysis of logs of Web traffic, we quantify how often the trusted-delegation assumption holds, and show that the remaining attack surface for CSRF attacks is very small. Finally, we have implemented our filtering algorithm as an extension of an existing client-side CSRF protection mechanism, and we show that our filtering does not break typical non-malicious cross-origin collaboration scenarios such as payment and single sign-on.

In summary, the contributions of this chapter are:

- The design of a novel client-side CSRF protection mechanism based on request filtering.
- A formalization of the algorithm, and formal evidence of the security of the algorithm under one specific assumption, the trusted-delegation assumption.
- An implementation of the countermeasure, and a validation of its compatibility with important Web scenarios broken by other state-of-the-art countermeasures.
- An experimental evaluation of the validity of the trusted-delegation assumption.

The remainder of the chapter is structured as follows. Section 3.2 explains the problem using both malicious and non-malicious scenarios. Section 3.3 discusses our request filtering mechanism. Section 3.4 introduces the formalization and results, followed by the implementation in Section 3.5. Section 3.6 experimentally evaluates the trusted-delegation assumption. Finally, Section 3.7 extensively discusses related work, followed by a brief conclusion (Section 3.8).

## 3.2 Cross-Origin HTTP Requests

The key challenge for a client-side CSRF prevention mechanism is to distinguish malicious from non-malicious cross-origin requests. This section illustrates the difficulty of this distinction by describing some attack scenarios and some important non-malicious scenarios that intrinsically rely on cross-origin requests.

### 3.2.1 Attack Scenarios

**A1. Classic CSRF.** Figure 3.1a shows a classic CSRF attack. In steps 1–4, the user establishes an authenticated session with site A, and later (steps 5–8) the user opens the malicious site E in another tab of the browser. The malicious page from E triggers a request to A (step 9), the browser considers this request to be part of the ongoing session with A and automatically adds the necessary authentication and session information. The browser internally maintains different *browsing contexts* for each origin it is interacting with. The shade of the browser-lifeline in the figure indicates the origin associated with the browsing context from which the outgoing request originates (also known as *the referrer*). Since the attack request originates from an E browsing context and goes to origin A, it is *cross-origin*.

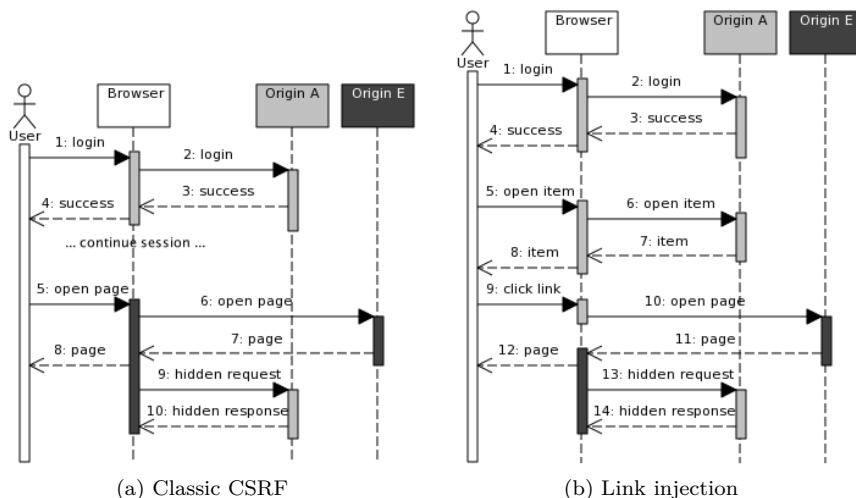


Figure 3.1: CSRF attack scenarios

For the attack to be successful, an authenticated session with A must exist when the user surfs to the malicious site E. The likelihood of success can be increased by making E content-related to A, for instance to attack a banking site, the attacker poses as a site offering financial advice.

**A2. Link Injection.** To further increase the likelihood of success, the attacker can inject links to E into the site A. Many sites, for instance social networking sites, allow users to generate content which is displayed to other users. For such a site A, the attacker creates a content item which contains a link to E. Figure

3.1b shows the resulting CSRF scenario, where A is a social networking site and E is the malicious site. The user logs into A (steps 1–4), opens the attacker injected content (steps 5–8), and clicks on the link to E (step 9) which launches the CSRF attack (step 13). Again, the attack request is cross-origin.

### 3.2.2 Non-Malicious Cross-Origin Scenarios

CSRF attack requests are cross-origin requests in an authenticated session. Hence, forbidding such requests is a secure countermeasure. Unfortunately, this also breaks many useful non-malicious scenarios. We illustrate two important ones.

**F1. Payment Provider.** Third-party payment providers such as PayPal or Visa 3D-secure offer payment services to a variety of sites on the Web. Figure 3.2a shows the scenario for PayPal’s *Buy Now* button. When a user clicks on this button, the browser sends a request to PayPal (step 2), that redirects the user to the payment page (step 4). When the user accepts the payment (step 7), the processing page redirects the browser to the dispatch page (step 10), that loads the landing page of the site that requested the payment (step 13).

Note that step 13 is a cross-origin request from PayPal to A in an authenticated session, for instance a shopping session in Web shop A.

**F2. Central Authentication.** The majority of interactive Websites require some form of authentication. As an alternative to each site using its own authentication mechanism, a single sign-on service (such as OpenID or Windows Live ID) provides a central point of authentication.

An example scenario for OpenID authentication using MyOpenID is shown in Figure 3.2b. The user chooses the authentication method (step 1), followed by a redirect from the site to the authentication provider (step 4). The authentication provider redirects the user to the login form (step 6). The user enters the required credentials, which are processed by the provider (step 10). After verification, the provider redirects the browser to the dispatching page (step 12), that redirects to the processing page on the original site (step 14). After processing the authentication result, a redirect loads the requested page on the original site (step 16). Again, note that step 16 is a cross-origin request in an authenticated session.

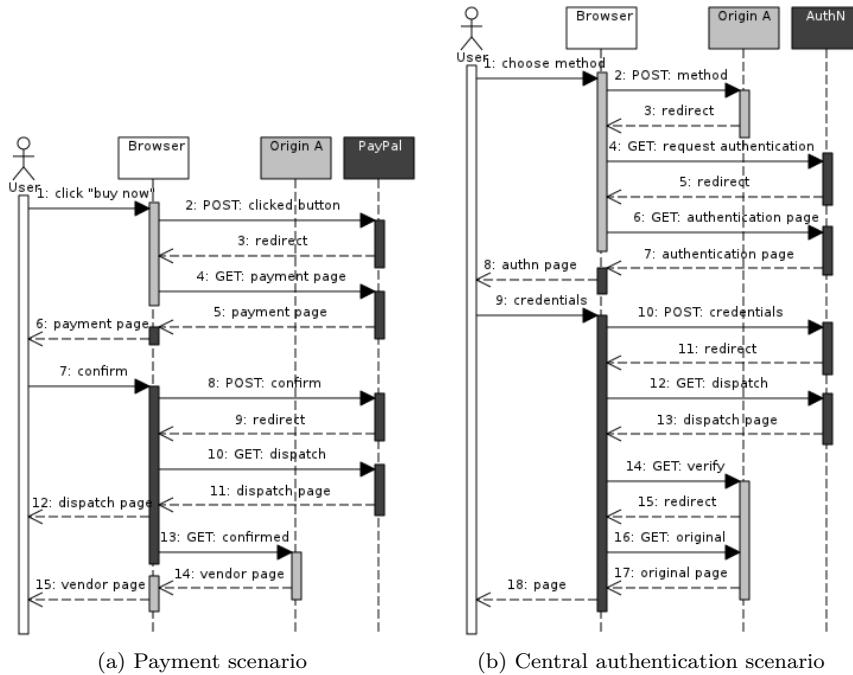


Figure 3.2: Non-malicious cross-origin scenarios

These two scenarios illustrate that mitigating CSRF attacks by preventing cross-origin requests in authenticated sessions breaks important and useful Web scenarios. Existing client-side countermeasures against CSRF attacks [64, 166, 211] either are incompatible with such scenarios or require user interaction for these cases.

### 3.3 Automatic and Precise Request Stripping

The core idea of our new countermeasure is the following: client-side state (i.e. session cookie headers and authentication headers) is stripped from all cross-origin requests, except for *expected* requests. A cross-origin request from origin A to B is *expected* if B previously (earlier in the browsing session) *delegated* to A. We say that B *delegates* to A if B either issues a POST request to A, or if B redirects to A using a URI that contains parameters.

The rationale behind this core idea is that (1) non-malicious collaboration scenarios follow this pattern, and (2) it is hard for an attacker to trick A into delegating to a site of the attacker: forcing A to do a POST or parametrized redirect to an evil site E requires the attacker to either identify a cross-site scripting (XSS) vulnerability in A, or to break into A’s Webserver. In both these cases, A has more serious problems than CSRF.

Obviously, a GET request from A to B is not considered a delegation, as it is very common for sites to issue GET requests to other sites, and as it is easy for an attacker to trick A into issuing such a GET request (see for instance attack scenario A2 in Section 3.2).

Unfortunately, the elaboration of this simple core idea is complicated somewhat by the existence of HTTP redirects. A Web server can respond to a request with a *redirect* response, indicating to the browser that it should resend the request elsewhere, for instance because the requested resource was moved. The browser will follow the redirect automatically, without user intervention. Redirects are used widely and for a variety of purposes, so we cannot ignore them. For instance, both non-malicious scenarios in Section 3.2 heavily depend on the use of redirects. In addition, attacker-controlled Websites can also use redirects in an attempt to bypass client-side CSRF protection. Akhawe et al. [10] discuss several examples of how attackers can use redirects to attack Web applications, including an attack against a CSRF countermeasure. Hence, correctly dealing with redirects is a key requirement for security.

The flowgraph in Figure 3.3 summarizes our filtering algorithm. For a given request, it determines what session state (cookies and authentication headers) the browser should attach to the request. The algorithm differentiates between simple requests and requests that are the result of a redirect.

**Simple Requests.** Simple requests that are not cross-origin, as well as expected cross-origin requests are handled as unprotected browsers handle them today. The browser automatically attaches the last known client-side state associated with the destination origin (point 1). The browser does not attach any state to non-expected cross-origin requests (point 3).

**Redirect Requests.** If a request is the consequence of a redirect response, then the algorithm determines if the redirect points to the origin where the response came from. If this is the case, the client-side state for the new request is limited to the client-side state known to the previous request (i.e. the request that triggered this redirect) (point 2). If the redirect points to another origin, then,

depending on whether this cross-origin request is expected or not, it either gets session-state automatically attached (point 1) or not (point 3).

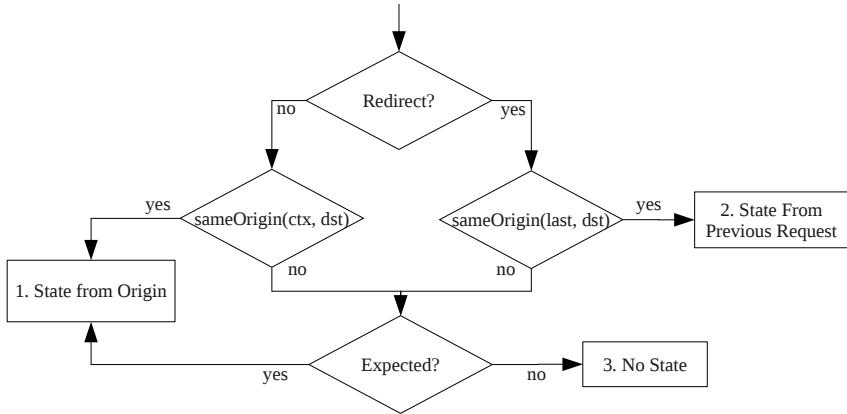


Figure 3.3: The request filtering algorithm

**When Is a Request Expected?** A key element of the algorithm is determining whether a request is *expected* or not. As discussed above, the intuition is: a cross-origin request from B to A is expected if and only if A first delegated to B by issuing a POST request to B, or by a parametrized redirect to B. Our algorithm stores such trusted delegations, and an assumption that we rely on (and that we refer to as the *trusted-delegation assumption*) is that sites will only perform such delegations to sites that they trust. In other words, a site A remains vulnerable to CSRF attacks from origins to which it delegates. Section 3.6 provides experimental evidence for the validity of this assumption.

The algorithm to decide whether a request is expected goes as follows. For a simple cross-origin request from site B to site A, a trusted delegation from site A to B needs to be present in the delegation store.

For a redirect request that redirects a request to origin Y (light gray) to another origin Z (dark gray) in a browsing context associated with some origin  $\alpha$ , the following rules apply.

1. First, if the destination (Z) equals the source (i.e.  $\alpha = Z$ ) (Figure 3.4a), then the request is expected if there is a trusted delegation from Z to Y in the delegation store. Indeed, Y is effectively doing a cross-origin request to Z by redirecting to Z. Since the browsing context has the same origin as the destination, it can be expected not to manipulate redirect requests to misrepresent source origins of redirects (cfr next case).

2. Alternatively, if the destination ( $Z$ ) is not equal to the source (i.e.  $\alpha \neq Z$ ) (Figure 3.4b), then the request is expected if there is a trusted delegation from  $Z$  to  $Y$  in the delegation store, since  $Y$  is effectively doing a cross-origin request to  $Z$ . Now, the browsing context might misrepresent source origins of redirects by including additional redirect hops (origin  $X$  (white) in Figure 3.4c). Hence, our decision to classify the request does not involve  $X$ .

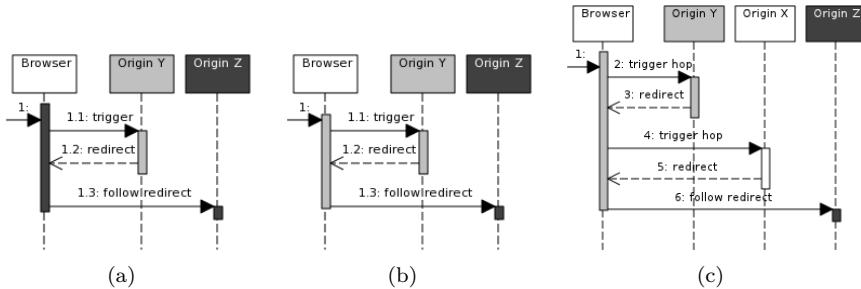


Figure 3.4: Complex cross-origin redirect scenarios

Finally, our algorithm imposes that expected cross-origin requests can only use the GET method and that only two origins can be involved in the request chain. These restrictions limit the potential power an attacker might have, even if the attacker successfully deceives the trusted-delegation mechanism.

**Mapping to Scenarios.** The reader can easily check that the algorithm blocks the attack scenarios from Section 3.2, and supports the non-malicious scenarios from that section. We discuss two of them in more detail.

In the PayPal scenario (Figure 3.2a), step 13 needs to re-use the state already established in step 2, which means that according to the algorithm, the request from PayPal to A should be expected. A trusted delegation happens in step 2, where a cross-origin POST is sent from origin A to PayPal. Hence the GET request in step 13 is considered expected and can use the state associated with origin A. Also note how the algorithm maintains the established session with PayPal throughout the scenario. The session is first established in step 3. Step 4 can use this session, because the redirect is an internal redirect on the PayPal origin. Step 8 can use the last known state for the PayPal origin and step 10 is yet another internal redirect.

In the link injection attack (Figure 3.1b), the attack happens in step 13 and is launched from origin E to site A. In this scenario, an explicit link between A

and E exists because of the link injected by the attacker. This link is however not a POST or parametrized redirect, so it is not a trusted delegation. This means that the request in step 10 is not considered to be expected, so it cannot access the previously established client-side state, and the attack is mitigated.

## 3.4 Formal Modeling and Checking

The design of Web security mechanisms is complex: the behaviour of (same-origin and cross-origin) browser requests, server responses and redirects, cookie and session management, as well as the often implicit threat models of Web security can lead to subtle security bugs in new features or countermeasures. In order to evaluate proposals for new Web mechanisms more rigorously, Akhawe et al. [10] have proposed a model of the Web infrastructure, formalized in Alloy.

The base model is some 2000 lines of Alloy source code, describing (1) the essential characteristics of browsers, Web servers, cookie management and the HTTP protocol, and (2) a collection of relevant threat models for the Web. The Alloy Analyzer – a bounded-scope model checker – can then produce counterexamples that violate intended security properties if they exist in a specified finite scope.

In this section, we briefly introduce Akhawe’s model and present our extensions to the model. We also discuss how the model was used to verify the absence of attack scenarios and the presence of functional scenarios.

### 3.4.1 Modeling our Countermeasure

The model of Akhawe et al. defines different principals, of which `GOOD` and `WEBATTACKER` are most relevant. `GOOD` represents an honest principal, who follows the rules imposed by the technical specifications. A `WEBATTACKER` is a malicious user who can control malicious Web servers, but has no extended networking capabilities.

The concept of `Origin` is used to differentiate between origins, which correspond to domains in the real world. An origin is linked with a server on the Web, that can be controlled by a principal. The browsing context, modeled as a `ScriptContext`, is also associated with an `origin`, that represents the origin of the currently loaded page, also known as the referrer.

A `ScriptContext` can be the source of a set of `HTTPTransaction` objects, which are a pair of an `HTTPRequest` and `HTTPResponse`. An HTTP request and

response are also associated with their remote destination origin. Both an HTTP request and response can have headers, where respectively the `CookieHeader` and `SetCookieHeader` are the most relevant ones. An HTTP request also has a `method`, such as GET or POST, and a `queryString`, representing URI parameters. An HTTP response has a `statusCode`, such as `c200` for a content result or `c302` for a redirect. Finally, an HTTP transaction has a cause, which can be none, such as the user opening a new page, a `RequestAPI`, such as a scripting API, or another `HTTPTransaction`, in case of a redirect.

To model our approach, we need to extend the model of Akhawe et al. to include (a) the accessible client-side state at a certain point in time, (b) the trusted delegation assumption and (c) our filtering algorithm. We discuss (a) and (b) in detail, but due to space constraints we omit the code for the filtering algorithm (c), which is simply a literal implementation of the algorithm discussed in Section 3.3.

**Client-side State.** We introduced a new signature `CSState` that represents a client-side state (Listing 3.1). Such a state is associated with an `Origin` and contains a set of `Cookie` objects. To associate a client-side state with a given request or response and a given point in time, we have opted to extend the `HTTPTransaction` from the original model into a `CSStateHTTPTransaction`. Such an extended transaction includes a `beforeState` and `afterState`, respectively representing the accessible client-side state at the time of sending the request and the updated client-side state after having received the response. The `afterState` is equal to the `beforeState`, with the potential addition of new cookies, set in the response.

---

```

1  sig CSState {
2    dst: Origin,
3    cookies: set Cookie
4  }
5
6  sig CSStateHTTPTransaction extends HTTPTransaction {
7    beforeState : CSState,
8    afterState : CSState
9  } {
10   //The after state of a transaction is equal to the before state + any additional cookies
11   // set in the response
12   beforeState.dst = afterState.dst
13   afterState.cookies = beforeState.cookies + (resp.headers & SetCookieHeader).thecookie
14
15   //The destination origin of the state must correspond to the transaction destination origin
16   beforeState.dst = req.host
17 }
```

---

Listing 3.1: Signatures representing our data in the model

**Trusted-delegation Assumption.** We model the trusted-delegation assumption as a fact, that honest servers do not send a POST or parametrized redirect to Web attackers (Listing 3.2).

---

```

1 fact TrustedDelegation {
2     all r : HTTPRequest | {
3         (r.method = POST || some (req.r).cause & CSStateHTTPTransaction)
4             &&
5             ((some (req.r).cause & CSStateHTTPTransaction
6                 && getPrincipalFromOrigin[(req.r).cause.req.host] in GOOD)
7                 || getPrincipalFromOrigin[transactions.(req.r).owner] in GOOD)
8             implies
9                 getPrincipalFromOrigin[r.host] not in WEBATTACKER
10            }
11        }

```

---

Listing 3.2: The fact modeling the trusted-delegation assumption

### 3.4.2 Using Model Checking for Security and Functionality

We formally define a CSRF attack as the possibility for a Web attacker (defined in the base model) to inject a request with at least one existing cookie attached to it (this cookie models the session/authentication information attached to requests) in a session between a user and an honest server (Listing 3.3).

---

```

1 pred CSRF[r : HTTPRequest] {
2     //Ensure that the request goes to an honest server
3     some getPrincipalFromOrigin[r.host]
4     getPrincipalFromOrigin[r.host] in GOOD
5
6     //Ensure that an attacker is involved in the request
7     some (WEBATTACKER.servers & involvedServers[req.r])
8     || getPrincipalFromOrigin[(transactions.(req.r)).owner] in WEBATTACKER
9
10    // Make sure that at least one cookie is present
11    some c : (r.headers & CookieHeader).thecookie | {
12        //Ensure that the cookie value is fresh
13        // (i.e. that it is not a renewed value in a redirect chain)
14        not c in ((req.r).*cause.resp.headers & SetCookieHeader).thecookie
15    }
16}

```

---

Listing 3.3: The predicate modeling a CSRF attack

We provided the Alloy Analyzer with a universe of at most 9 HTTP events and where an attacker can control up to 3 origins and servers (a similar size as used in [10]). In such a universe, no examples of an attacker injecting a request

through the user’s browser were found. This gives strong assurance that the countermeasure does indeed protect against CSRF under the trusted delegation assumption.

We also modeled the non-malicious scenarios from Section 3.2, and the Alloy Analyzer reports that these scenarios are indeed permitted. From this, we can also conclude that our extension of the base model is consistent.

Space limitations do not permit us to discuss the detailed scenarios present in our model, but the interested reader can find the complete model available for download at [68].

## 3.5 Implementation

We have implemented our request filtering algorithm in a proof-of-concept add-on for the Firefox browser, and used this implementation to conduct an extensive practical evaluation. First, we have created simulations for both the common attack scenarios as well as for the two functional scenarios discussed in the paper (third party payment and centralized authentication), and verified that they behaved as expected.

Second, in addition to these simulated scenarios, we have verified that the prototype supports actual instances of these scenarios, such as for example the use of MyOpenID authentication on sourceforge.net.

Table 3.1: CSRF benchmark

	Test scenarios	Result
HTML	29 cross-origin test scenarios	✓
CSS	12 cross-origin test scenarios	✓
ECMAScript	9 cross-origin test scenarios	✓
Redirects	20 cross-origin redirect scenarios	✓

Third, we have constructed and performed a CSRF benchmark,<sup>3</sup> consisting of 70 CSRF attack scenarios to evaluate the effectiveness of our CSRF prevention technique (see Table 3.1). These scenarios are the result of a CSRF-specific study of the HTTP protocol, the HTML specification and the CSS markup language to examine their cross-origin traffic capabilities, and include complex redirect scenarios as well. Our implementation has been evaluated against each of these scenarios, and our prototype passed all tests successfully.

---

<sup>3</sup>The benchmark can be applied to other client-side solutions as well, and is downloadable at [68].

Table 3.2: Analysis of the trusted-delegation assumption in a real-life data set of 4,729,217 HTTP requests

	# requests	POST	redir.
Third party service mashups	29,282 (52.95%)	5,321	23,961
<i>Advertisement services</i>	22,343 (40.40%)	1,987	20,356
<i>Gadget provider services</i>	2,879 (5.21%)	2,757	122
<i>Tracking services</i>	2,864 (5.18%)	411	2,453
<i>(appspot, mochibot, gmodules, ...)</i>			
<i>Single Sign-On services</i>	1,156 (2.09%)	137	1,019
<i>(Shibboleth, Live ID, OpenId, ...)</i>			
<i>3rd party payment services</i>	27 (0.05%)	19	8
<i>(Paypal, Ogone)</i>			
<i>Content sharing services</i>	13 (0.02%)	10	3
<i>(addtoany, sharethis, ...)</i>			
Multi-origin Websites	13,973 (25.27%)	198	13,775
Content aggregators	8,276 (14.97%)	0	8,276
<i>Feeds</i>	4,857 (8.78%)	0	4,857
<i>(RSS feeds, News aggregators, ...)</i>			
<i>Redirecting search engines</i>	3,344 (6.05%)	0	3,344
<i>(Google, Comicranks, Ohnorobot)</i>			
<i>Document repositories</i>	75 (0.14%)	0	75
<i>(ACM digital library, dx.doi.org, ...)</i>			
False positives	1,215 (2.20%)	12	1,203
<i>(wireless network access gateways)</i>			
URL shorteners	759 (1.37%)	0	759
<i>(gravatar, bit.ly, tinyurl, ...)</i>			
Others	1,795 (3.24%)	302	1,493
<i>(unclassified)</i>			
<b>Total No. of 3rd party delegation initiators</b>	<b>55,300</b> <b>(100%)</b>	<b>5,833</b>	<b>49,467</b>

The prototype, the scenario simulations and the CSRF benchmark suite are available for download [68].

## 3.6 Evaluating the Trusted-Delegation Assumption

Our countermeasure drastically reduces the attack surface for CSRF attacks. Without CSRF countermeasures in place, an origin can be attacked by any other origin on the Web. With our countermeasure, an origin can only be attacked by another origin to which it has delegated control explicitly by means of a cross-origin POST or redirect. We have already argued in Section 3.3 that it is difficult for an attacker to cause unintended delegations. In this section, we measure the remaining attack surface experimentally.

We conducted an extensive traffic analysis using a real-life data set of 4,729,217 HTTP requests, collected from 50 unique users over a period of 10 weeks. The analysis revealed that 1.17% of the 4.7 million requests are treated as delegations in our approach. We manually analyzed all these 55,300 requests, and classified them in the interaction categories summarized in Table 3.2.

For each of the categories, we discuss the resulting attack surface:

**Third party service mashups.** This category consists of various third party services that can be integrated in other Websites. Except for the single sign-on services, this is typically done by script inclusion, after which the included script can launch a sequence of cross-origin GET and/or POST requests towards offered AJAX APIs. In addition, the service providers themselves often use cross-origin redirects for further delegation towards content delivery networks.

As a consequence, the origin A including the third-party service S becomes vulnerable to CSRF attacks from S. This attack surface is unimportant, as in these scenarios, S can already attack A through script inclusion, a more powerful attack than CSRF.

In addition, advertisement service providers P that further redirect to content delivery services D are vulnerable to CSRF attacks from D whenever a user clicks an advertisement. Again, this attack surface is unimportant: the delegation from P to D correctly reflects a level of trust that P has in D, and P and D will typically have a legal contract or SLA in place.

**Multi-origin Websites.** Quite a number of larger companies and organizations have Websites spanning multiple origins (such as *live.com - microsoft.com* and *google.be - google.com*). Cross-origin POST requests and redirects between these origins make it possible for such origins to attack each other. For instance, *google.be* could attack *google.com*. Again, this attack surface is unimportant, as all origins of such a multi-origin Website belong to a single organization.

**Content aggregators.** Content aggregators collect searchable content and redirect end-users towards a specific content provider. For news feeds and document repositories (such as the ACM digital library), the set of content providers is typically stable and trusted by the content aggregator, and therefore again a negligible attack vector.

Redirecting search engines register the fact that a Web user is following a link, before redirecting the Web user to the landing page (e.g. as Google does for logged in users). Since the entries in the search repository come from all over the Web, our CSRF countermeasure provides little protection for such search engines. Our analysis identified 4 such origins in the data set: *google.be*, *google.com*, *comicrank.com*, and *ohnorobot.com*.

**False positives.** Some fraction of the cross-origin requests are caused by network access gateways (e.g. on public Wifi) that intercept and reroute

requests towards a payment gateway. Since such devices have man-in-the-middle capabilities, and hence more attack power than CSRF attacks, the resulting attack surface is again negligible.

**URL shorteners.** To ease URL sharing, URL shorteners transform a shortened URL into a preconfigured URL via a redirect. Since such URL shortening services are open, an attacker can easily control a new redirect target. The effect is similar to the redirecting search engines; URL shorteners are essentially left unprotected by our countermeasure. Our analysis identified 6 such services in the data set: *bit.ly*, *gravatar.com*, *post.ly*, *tiny.cc*, *tinyurl.com*, and *twitpic.com*.

**Others (unclassified)** For some of the requests in our data set, the origins involved in the request were no longer online, or the (partially anonymized) data did not contain sufficient information to reconstruct what was happening, and we were unable to classify or further investigate these requests.

In summary, our experimental analysis shows that the trusted delegation assumption is realistic. Only 10 out of 23.592 origins (i.e. 0.0042% of the examined origins) – the redirecting search engines and the URL shorteners – perform delegations to arbitrary other origins. They are left unprotected by our countermeasure. But the overwhelming majority of origins delegates (in our precise technical sense, i.e. using cross-origin POST or redirect) only to other origins with whom they have a trust relationship.

### 3.7 Related Work

The most straightforward protection technique against CSRF attacks is server-side mitigation via validation tokens [47, 144]. In this approach, Web forms are augmented with a server-generated, unique validation token (e.g. embedded as a hidden field in the form), and at form submission the server checks the validity of the token before executing the requested action. At the client-side, validation tokens are protected from cross-origin attackers by the same-origin-policy, distinguishing them from session cookies or authentication credentials that are automatically attached to any outgoing request. Such a token based approach can be offered as part of the Web application framework [208, 83], as a server-side library or filter [241], or as a server-side proxy [144].

Recently, the `Origin` header has been proposed as a new server-side countermeasure [26, 24]. With the `Origin` header, clients unambiguously inform

the server about the origin of the request (or the absence of it) in a more privacy-friendly way than the `Referer` header. Based on this origin information, the server can safely decide whether or not to accept the request. In follow-up work, the `Origin` header has been improved, after a formal evaluation revealed a previously unknown vulnerability [10]. The Alloy model used in this evaluation also formed the basis for the formal validation of our presented technique in Section 3.4.

Unfortunately, the adoption rate of these server-side protection mechanisms is slow, giving momentum to client-side mitigation techniques as important (but hopefully transitional) solutions. In the next paragraphs, we will discuss the client-side proxy RequestRodeo, as well as four mature and popular browser addons (CsFire, NoScript ABE, RequestPolicy, and CSD<sup>4</sup>). In addition, we will evaluate how well the browser addons enable the functional scenarios and protect the user against the attack scenarios discussed in this chapter (see Table 3.3).

RequestRodeo [143] is a client-side proxy proposed by Johns and Winter. The proxy applies a client-side token-based approach to tie requests to the correct source origin. In case a valid token is lacking for an outgoing request, the request is considered suspicious and gets stripped of cookies and HTTP `authorization` headers. RequestRodeo lies at the basis of most of the client-side CSRF solutions [64, 165, 166, 211, 264], but because of the choice of a proxy, RequestRodeo often lacks context information, and the rewriting technique on raw responses does not scale well in a Web 2.0 world.

CsFire [64] extends the work of Maes et al. [163], and strips cookies and HTTP `authorization` headers from a cross-origin request. The advantage of stripping is that there are no side-effects for cross-origin requests that do not require credentials in the first place. CsFire operates autonomously by using a default client policy which is extended by centrally distributed policy rules. Additionally, CsFire supports users creating custom policy rules, which can be used to blacklist or whitelist certain traffic patterns. Without a central or user-supplied whitelist, CsFire does not support the payment and central authentication scenario.

To this extent, we plan to integrate the approach presented in this chapter to the CsFire Mozilla Add-On distribution in the near future.

NoScript ABE [166], or Application Boundary Enforcer, restricts an application within its origin, which effectively strips credentials from cross-origin requests, unless specified otherwise. The default ABE policy only prevents CSRF attacks from the internet to an intranet page. The user can add specific policies, such

---

<sup>4</sup>Since the client-side detection technique described in [220] is not available for download, the evaluation is done based on the description in the paper.

Table 3.3: Evaluation of browser-addons

	Functional scenarios		Attack scenarios	
	F1. Payment Provider	F2. Central Authentication	A1. Classic CSRF	A2. Link Injection
CsFire [64]	✗	✗	✓	✓
NoScript ABE [166] <sup>a</sup>	✗	✗	✓	✓
RequestPolicy [211]	▷ <sup>b</sup>	▷ <sup>b</sup>	✓ <sup>c</sup>	✓ <sup>c</sup>
Client-side Detection [220]	✗	✗	✓	✓
<b>Our Approach</b>	✓	✓	✓	✓

<sup>a</sup> ABE configured as described in [102].

<sup>b</sup> Requires interactive feedback from end-user to make the decision.

<sup>c</sup> Requests are blocked instead of stripped, impacting the end-user experience.

as a CsFire-alike stripping policy [102], or a site-specific blacklist or whitelist. If configured with [102], ABE successfully blocks the three attack scenarios, but disables the payment and central authentication scenario.

RequestPolicy [211] protects against CSRF by blocking all cross-origin requests. In contrast to stripping credentials, blocking a request can have a very noticeable effect on the user experience. When detecting a cross-origin redirect, RequestPolicy injects an intermediate page where the user can explicitly allow the redirect. RequestPolicy also includes a predefined whitelist of hosts that are allowed to send cross-origin requests to each other. Users can add exceptions to the policy using a whitelist. RequestPolicy successfully blocks the three attack scenarios (by blocking instead of stripping all cross-origin requests) and requires interactive end-user feedback to enable the payment and central authentication scenario.

Finally, in contrast to the CSRF *prevention* techniques discussed in this chapter, Shahriar and Zulkernine proposes a client-side *detection* technique for CSRF [220]. In their approach, malicious and benign cross-origin requests are distinguished from each other based on the existence and visibility of the submitted form or link in the originating page, as well as on the visibility of the target. In addition, the expected content type of the response is taken into account to detect false negatives during execution. Although the visibility check closely approximates the end-user intent, their technique fails to support the script inclusions of third party service mashups as discussed in Section 3.6. Moreover, without taking into account the delegation requests, expected redirect requests (as defined in Section 3.3) will be falsely detected as CSRF attacks, although these requests are crucial enablers for the payment and central authentication scenario.

## 3.8 Conclusion

We have proposed a novel technique for protecting at client-side against CSRF attacks. The main novelty with respect to existing client-side countermeasures is the good trade-off between security and compatibility: existing countermeasures break important Web scenarios such as third-party payment and single-sign-on, whereas our countermeasure can permit them.



## Chapter 4

# Preventing Session Fixation Attacks in the Browser

The paper in this chapter introduces *Serene*, a browser add-on that protects against session fixation attacks, which enable an *unauthorized transfer of the session*. Session fixation attacks can be executed through 6 distinct attack vectors, making the *forum poster*, *Web attacker*, *related domain attacker* and *passive/active network attacker* relevant threat models. Serene essentially keeps track of session identifiers issued by a Web application, and only allows the use of these session identifiers, thereby preventing the use of a potentially fixated session identifier. The evaluation focuses on compatibility with the Alexa top 1,000,000 sites, showing that Serene fully preserves 95.14% of functionality, while it is able to autonomously protect 83.43% of applications. This paper was presented at the *12th International IFIP Conference on Distributed Applications and Interoperable Systems* (DAIS 2012) [74].<sup>1</sup>

While session fixation attacks also rank highly in the OWASP Top 10 [257] and the CWE/SANS Top 25 [172], they are often underestimated. Typical defenses against session fixation are deployed at the server-side, and depend on cooperation of the developer. Serene is the first client-side mitigation technique against session fixation attacks, and works both for cookie-based and parameter-based session management systems. Serene's main challenge lies in both maximizing the scope of its protection and minimizing its interference with Web applications.

---

<sup>1</sup>Philippe De Ryck and Nick Nikiforakis took the lead on this paper. Philippe focused on implementation and evaluation.

Serene’s design consists of two main components, a heuristic algorithm to identify the session identifiers in a set of cookies or parameters, and the protection mechanism that effectively prevents the fixation of a session identifier by an attacker. This design enables the maintaining of a high degree of compatibility, even in the future Web, as the heuristics algorithm most likely requires finetuning as the Web further evolves. Follow-up research by Calzavara et al. [49] focuses exactly on the identification of session identifiers, and proposes a semi-automatic machine learning technique to construct a so-called *Golden Set* for 70 popular Web applications. Such a golden set defines the set of cookies that serve as the actual *authentication token*. Based on these golden sets, the authors evaluated several mitigation techniques that depend on the detection of session identifiers, including Serene and SessionShield [185], which served as an inspiration for Serene. Their results show that Serene’s heuristic algorithm improves SessionShield’s false positive rate from 105 to 37 out of 327 cookies. However, Serene incurs a larger false negative rate compared to SessionShield (55 to 8 out of 103).

In hindsight, we can conclude that Serene’s heuristic algorithm causes little compatibility issues, but can still be improved to cover more cookies that are part of the authentication token, a need we already correctly assessed in the initial paper, and remains a major challenge in the modern Web. The golden sets proposed by Calzavara et al. are extremely valuable in showing the complexity of authentication tokens. They aptly highlight the difficulty to establish the exact composition of an authentication token from the client side, which still requires the need to resort to manual or semi-manual processes requiring a significant amount of human investment. Hopefully, future work will either bring a fully automated detection mechanism for authentication tokens, albeit that a server-driven policy to mark certain tokens as authentication tokens may be more feasible.

# Serene: Self-Reliant Client-side Protection against Session Fixation

**Abstract** *The Web is the most wide-spread and de facto distributed platform, with a plethora of valuable applications and services. Building stateful services on the Web requires a session mechanism that keeps track of server-side session state, such as authentication data. These sessions are an attractive attacker target, since taking over an authenticated session fully compromises the user’s account. This chapter focuses on session fixation, where an attacker forces the user to use the attacker’s session, allowing the attacker to take over the session after authentication.*

We present Serene, a self-reliant client-side countermeasure that protects the user from session fixation attacks, regardless of the security provisions – or lack thereof – of a Web application. By specifically protecting session identifiers from fixation and not interfering with other cookies or parameters, Serene is able to autonomously protect a large majority of Web applications, without being disruptive towards legitimate functionality. We experimentally validate these claims with a large scale study of Alexa’s top one million sites, illustrating both Serene’s large coverage (83.43%) and compatibility (95.55%).

## 4.1 Introduction

In the past few years, the security community has witnessed a shift in attacks originating from malicious individuals and the organized criminal underground. Attacks usually targeting the server-side of the Internet (e.g. Web, Mail and FTP servers) are now conducted on the client-side, targeting the site, the user’s browser or even the user himself. This phenomenon can be ascribed to the enormous expansion of Web sites and Web applications, which currently almost monopolize a user’s online activities. A substantial fraction of these attacks targets a Web application’s session management, the cornerstone of any stateful Web application. Session management enables building stateful applications on top of a stateless protocol (HTTP), by grouping multiple related requests together into a session. Each session is assigned a unique identifier and can keep track of session-specific data, such as preferences, user information or authentication state. Sessions are typically maintained by cookies, part of the HTTP headers, or parameters, embedded in the content.

One well-known session attack is *session fixation*. In a session fixation attack, the attacker establishes a session between him and the target application, and subsequently forces this session into the user’s browser. Any action taken by the user within the application is associated with the user’s session, which is in this case identical to the attacker’s session. For example, if the user authenticates herself to the application, the session remembers the user’s information and authentication state. In case of a session fixation attack, the attacker shares the same session, allowing him to perform actions in the user’s name. Session fixation is ranked third in the OWASP top 10 of Web application security risks, and is assigned a prevalence of *common* [258].

An adequate, widely available by-design mitigation technique for session fixation is to issue a new (thus non-fixed) session identifier whenever the privilege level of a user changes, for example from unauthenticated to authenticated. Unfortunately, studies have shown that security guidelines are not applied as widespread as one would hope or expect [219, 266], thus leaving the user vulnerable for potential session fixation attacks.

We present Serene, a self-reliant client-side countermeasure against session fixation attacks. Serene is compatible with applications using both cookie-based and/or parameter-based session management. The main idea behind Serene is to prevent the browser from sending fixated session identifiers through cookies, and to prevent the use of fixated session identifiers through parameters embedded in the pages’ contents. To distinguish session identifiers from other cookies or parameters, we present an elementary algorithm that supports a large majority of sites, but still maintains a very low false positive rate. To validate our identification algorithm and test our prototype implementation, we conducted a large scale study of Alexa’s top one million sites, showing both the wide range of support and the compatibility of Serene.

The remainder of this chapter is organized as follows: Section 4.2 introduces parameter-based and cookie-based session management techniques. Section 4.3 focuses entirely on session fixation, including different attack vectors, current countermeasures and a real-life example attack scenario. Section 4.4 presents Serene, our client-side countermeasure against session fixation attacks. We extensively validate Serene using the Alexa top one million (Section 4.5). Finally, we discuss difficulties with re-using existing previous work, as well as potential improvements (Section 4.6). We conclude the paper in Section 4.7.

## 4.2 Session Management

Virtually every non-static Web application embodies stateful behavior such as identifying individual users, enforcing access control rules and distinguishing simultaneously submitted requests. Due to the stateless nature of HTTP, this stateful behavior is enabled on top of HTTP by introducing sessions. Sessions link multiple requests from the same client together and allow stateful information to be accessed and updated during the course of that session.

The de facto implementation of sessions consists of a server-side stored session state, for which the server generates a random, unique session identifier (SID) [185]. The client is instructed to include the assigned SID with every request, allowing the server to link multiple requests from this client to the same session. There are two common ways a Web application can instruct the client to include a SID: cookies and parameters. We discuss both approaches separately in the remainder of this section.

### 4.2.1 Cookies as Session Identifiers

Cookies are key/value pairs belonging to the domain that sets them, potentially extended by certain options (e.g. *Path*, *Secure*, etc.). The browser keeps track of cookies in the so-called cookie jar. When the browser sends a request to a certain domain, it attaches all known cookies for that domain using the `Cookie` request header. Traditionally, cookies are set by the server using either the `Set-Cookie` response header, by embedding a `Set-Cookie` meta tag in the body or by including JavaScript that sets a cookie when executed by the browser.

Cookies typically belong to the domain that sets them (e.g. `www.example.com`), but using the *Domain* option, they can also be bound to a parent domain (e.g. `.example.com`), in which case they belong to all subdomains of `example.com`. This feature is often used to share the same cookies among different parts of an application (e.g. `login.bank.com` and `payments.bank.com`). Setting the *Domain* option to a top-level domain (e.g. `.com`) is not allowed.

Implementing session management using cookies is straightforward: a session is typically created by the server after receiving the first (cookieless) request, and the generated SID is attached as a cookie to the response. The browser stores the cookie containing the SID and attaches it to every request going to this specific domain. Upon receiving a request containing a cookie with a SID, the server can link the request to the associated session. A new SID can easily be assigned by sending the client a new cookie with the same name but a different value, which overwrites the old value.

### 4.2.2 Parameters as Session Identifiers

Since not all clients support cookies, or cookie support can be explicitly disabled by the user, an alternative approach is to include the SID as a parameter in every request to the server. Examples are to include the SID as a parameter in the URL of a link, or as a hidden field in a form element. Maintaining a session this way requires the server to ensure that all URLs pointing to its own domain contain the SID of the associated session. When the user opens a URL with an embedded SID (e.g. by clicking on a link), the browser sends a request containing the embedded SID, allowing the server to extract the SID and link the request to the associated session.

Popular Web frameworks offer embedded support for session management, which includes both cookie-based and parameter-based session management. Sites running on top of such a framework can easily support the parameter-based fallback mode if desired.

### 4.2.3 Attacks on Session Management

Attacks on session management are popular, since they offer a high reward for the attacker. Successful attacks can involve the attacker making specific requests in the user's name, or an attacker having full control over the user's session, allowing him to access all information and perform all actions available to the user. Concrete attack examples include session hijacking and cross-site request forgery (CSRF). Existing work proposes specific client-side countermeasures to prevent both session hijacking [148, 185] and CSRF attacks [65, 211].

In this chapter we focus on session fixation and propose a client-side countermeasure against this attack. To the best of our knowledge, Serene is the first concrete proposal for client-side protection against session fixation attacks.

## 4.3 Session Fixation

The goal of a session fixation attack is to gain control over a session of an authenticated user, thus giving the attacker full access to the target application with the user's privileges. To reach this goal, the attacker will force the user to use a session accessible to the attacker, by fixating a known SID in the user's browser before authentication. When the user has successfully authenticated, the attacker can contact the target application with the same

SID, thus impersonating the authenticated user. Figure 4.1a shows a session fixation attack on a parameter-based session management system.

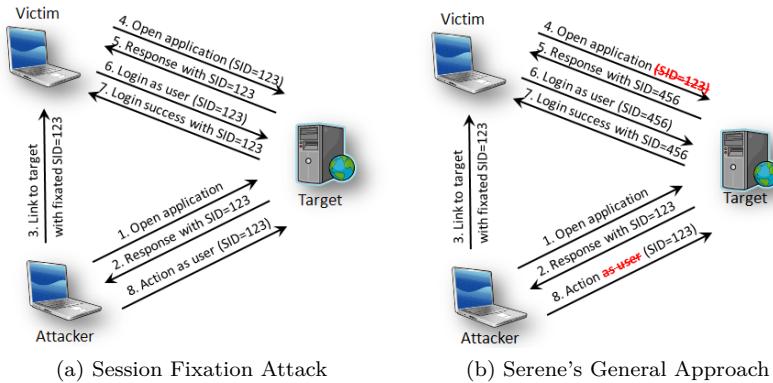


Figure 4.1: Schematic representation of a session fixation attack (left) and Serene’s general protection approach (right)

To launch a session fixation attack against a vulnerable application, the attacker needs to be able to fixate the SID in the user’s browser. The attacker can gain this capability by launching the attack from a Web site, the so-called Web attacker [27], or through an out-of-band channel, such as email or instant messaging. In the remainder of this section, we discuss ways for an attacker to perform session fixation attacks on parameter-based and cookie-based session management systems. Additionally, we discuss currently available protection mechanisms against session fixation attacks, followed by a real-life attack scenario discovered during the writing of this chapter.

### 4.3.1 Parameter-Based Session Fixation

**A1. Links.** The simplest attack vector is to get the user to go to the target site through a crafted link, that contains a SID embedded as a URL parameter (e.g. `<a href="http://target.example.com?SESSID=A1B2C3D4E5">Follow me</a>`). There are numerous ways of tricking the user into following a crafted link. One example are external channels, such as email messages or instant messaging, where a user is simply asked to open a link. An attacker can also place a link on an unrelated site, such as an attacker-controlled site or a site permitting user content to be posted, such as a social networking site, a forum, etc. Finally, an attacker can also inject a link containing a SID directly in the target site, making it blend in with the rest of the page contents.

**A2. Script Execution.** If the attacker can execute a malicious script within the origin of the target site, he can easily launch a session fixation attack by replacing the valid embedded SIDs with fixated SIDs.

Script execution privileges can be gained legitimately (e.g. an included advertisement), or unintentionally (e.g. through cross-site scripting (XSS)). Unfortunately, vulnerabilities giving attackers script execution privileges are very common, as indicated by the high-ranked spot in both the OWASP top 10 Web application security risks [258]. Additionally, attackers can legitimately gain script execution privileges in numerous ways [8, 237].

### 4.3.2 Cookie-Based Session Fixation

**A3. Script Execution.** Similar to parameter-based session fixation (attack vector A2), cookie-based session management is susceptible to session fixation if the attacker gains script execution privileges. The attacker can simply set a fixated SID as a cookie through the `document.cookie` property. Note that a site with an XSS vulnerability that allows cookie-based session fixation, is not necessarily susceptible to other attacks, such as session hijacking [266].

**A4. Meta-tags.** The `http-equiv` attribute supported by HTML meta tags enables several header-like instructions, such as setting cookies. Placing the following code in an HTML page results in the creation of a cookie with name *foo* and value *bar*: `<meta http-equiv="Set-Cookie" content="foo=bar; Path=/;" />`. By injecting such a tag in the target site, an attacker can easily fixate a SID in a cookie.

Meta tags are typically included in the header of a page, but an investigation of major browsers shows that meta tags found in the page's body are also honored. Additionally, some browsers also process dynamically included meta tags (e.g. from JavaScript using the `appendChild` operation). Similar to script execution, this attack vector can be exploited through legitimate means (e.g. an included advertisement) or through an injection vulnerability. Note that for meta tag injection, it suffices that the injection vulnerability allows the injection of an HTML tag. A full-scale cross-site scripting vulnerability is not required.

**A5. Headers.** The possibility for an attacker to inject headers into the HTTP response allows him to use the `Set-Cookie` header to fixate a cookie-based SID. Header injection [161] is typically caused by a target site including unsanitized input in header values, or by a parsing vulnerability in a browser or proxy system. Due to the large-scale impact of such a vulnerability in a Web

framework, language or browser, these vulnerabilities are typically immediately fixed, making header injection an unlikely attack vector.

**A6. Subdomains.** As discussed before, cookies set from a subdomain can apply to other subdomains as well, using the *Domain* option. This feature becomes problematic when not all subdomains belong to the same entity, and the attacker controls one of them.

An obvious way for the attacker to attack a target site sharing the same parent domain is to set a cookie for the parent domain through an HTTP header. If the attacker fully controls this application, setting cookies through HTTP headers is trivial. In cases where the attacker has no control over the headers (e.g. limited hosting with only static pages), he can achieve the same goal by including a meta tag in one of his pages or by setting a cookie from JavaScript through the `document.cookie` property. At the end of this section, we discuss a real-life session fixation attack where the attacker is able to execute scripts in a subdomain, without having control over the headers.

### 4.3.3 Current Countermeasures

Session fixation attacks have been known for some time, and adequate server-side protection techniques are available. Unfortunately, these protection techniques are not always deployed, leaving the user vulnerable to session fixation attacks. We discuss the most important and effective countermeasures below.

Session fixation can easily be addressed during the development phase of a Web application, by generating a new session identifier whenever the privilege level of a user changes, for example from an unauthenticated state to an authenticated state. This approach foils any session fixation attacks aimed at obtaining an authenticated session. Even if an attacker forces a session identifier on a user, the session identifier will be overwritten by a newly generated one after authentication. Since the new SID differs from the fixated one, the authenticated user is never linked to the fixated session. This approach works both for cookie-based and parameter-based session management. Most Web frameworks explicitly support the regeneration of SIDs, but require the developer to enable it or explicitly trigger it by calling a function.

Instead of focusing on the value of the SID, several approaches aim to generally protect cookies. One example is the *HttpOnly* option that can be added to a cookie, preventing JavaScript from reading that cookie [266], foiling traditional session hijacking attacks. Recently, browsers started preventing an HttpOnly cookie to be overwritten from JavaScript, thus severely limiting the window of

opportunity for a session fixation attack using attack vectors *A3*, *A4* and *A5* (i.e. fixation can only happen before the user received a SID from the server). As indicated by other studies, the use of HttpOnly is still fairly limited [219, 266].

Bortz et al. [41] propose to limit cookies to their origin, thus preventing the sharing of cookies across subdomains. Origin cookies can effectively prevent session fixation attacks from subdomains, if both browsers and applications implement and deploy this feature.

Furthermore, Johns et al. [141] have proposed two more server-side solutions for combating session fixation attacks against cookie-based session management. One consists of instrumenting the underlying Web framework, to automatically regenerate the SID when an authentication process is detected. In a second approach, they propose a server-side proxy that maintains its own SID and couples it to the target site's SID. Renewing the proxy SID after a detected authentication process prevents session fixation attacks. Both approaches do not require modifications to either the Web framework or the protected site, but depend on initial training or configuration to identify the authentication process.

#### 4.3.4 Example Attack Scenario

While researching possible attacks and defenses connected with session fixation, we encountered a two-step session fixation attack on Weebly, a Web 2.0 site builder boasting a userbase of more than 8 million people. When registering on [www.weebly.com](http://www.weebly.com), a dedicated subdomain is assigned to the user (e.g. `alice.weebly.com`). The user can subsequently create her site using a combination of drag-and-drop elements as well as custom HTML, which allows a user to write arbitrary HTML and JavaScript code on her page.

Weebly's cookie configuration settings do not allow a page on a subdomain to steal cookies from sibling domains or the main [www.weebly.com](http://www.weebly.com) domain. This effectively prevents session hijacking attacks that attempt to steal cookie-based SIDs through JavaScript. As discussed in attack vector *A6*, JavaScript is allowed to create cookies with the `domain` option set to `.weebly.com`. This operates as a subdomain-wildcard, instructing the browser to send this cookie to all subdomains of `weebly.com`. Thus, an attacker can now set a cookie that will be sent to [www.weebly.com](http://www.weebly.com) when the user visits Weebly's home page.

At the same time, we noticed that if a user presents a session identifier to Weebly's login screen and successfully logs-in, Weebly maintains the same identifier. These two "features" provide all the necessary ingredients for a session fixation attack. If a user is lured into visiting a malicious site hosted

on a subdomain of `weebly.com`, the attacker can fixate a cookie-based SID valid on `www.weebly.com` in the user's browser. Once the user authenticates to Weebly, the attacker can take full control over the user's session. All the attacker needs to do, is to poll one of Weebly's authenticated pages until he is recognized as the logged-in victim.

We have reported the vulnerability and way of exploitation to Weebly's staff, who verified and corrected the issue as of November 2011. When a Weebly user now authenticates on `www.weebly.com`, the SID will be renewed.

## 4.4 Client-Side Protection against Session Fixation

Even though adequate protection techniques and countermeasures exist, Web applications do not implement them, leaving the user vulnerable. In this section, we present Serene, a client-side countermeasure against session fixation, that will protect the user against session fixation attacks, regardless of the security precautions taken by any target application. Serene does not depend on the user to make security decisions, since the user cannot be expected to have the expertise nor the time to make a decision for each application or request.

We present our countermeasure in several steps. First, we discuss the general idea to protect the session against session fixation attacks. Next, we elaborate on a few peculiarities with parameter-based session management, followed by our algorithm to identify SIDs from the collection of cookie and parameter key/value pairs. Finally, we discuss the prototype implementation as a Firefox add-on.

### 4.4.1 General Approach

The general approach of protecting against session fixation attacks is to prevent potentially fixated SIDs from being sent to the server. If a fixated SID never reaches the server, session fixation is simply not possible. This general approach, as depicted in Figure 4.1b, is applicable for both cookie-based and parameter-based session management.

For cookie-based session management, Serene keeps track of all legitimately set cookies that contain a SID in an internal database. These cookies are found in the `Set-Cookie` header, part of the response. Next, Serene scans each outgoing request for attached cookies, by investigating the `Cookie` header. Any attached cookies that contain a SID must also appear in the internal database of legitimate SIDs for the target site. In case a cookie containing a SID is not

present in the internal database, it is stripped from the outgoing request. The reasoning behind this approach builds on the fact that it is very unlikely for a server-generated session identifier to be set in other ways than through a header. In Section 4.5, we offer substantial experimental evidence to support this claim.

Mapping the attack vectors discussed earlier to this approach shows that cookies containing a SID set through JavaScript or meta tags are not known in the internal database, and thus not sent to the server. This holds both for the single domains as well as for the subdomain case. The only remaining attack vector is through header injection in a single domain, a highly unlikely scenario, or through header injection from a subdomain, discussed in Section 4.6.

To protect applications with parameter-based session management, incoming pages are scanned for embedded SIDs (e.g. in URLs or as hidden form fields) that identify a session within the domain sending the response. Similar to cookies, these SIDs are stored in the internal database. Outgoing requests are also scanned for any embedded SIDs (e.g. in the URL or in the POST body), and SIDs not present in the database are removed from the request.

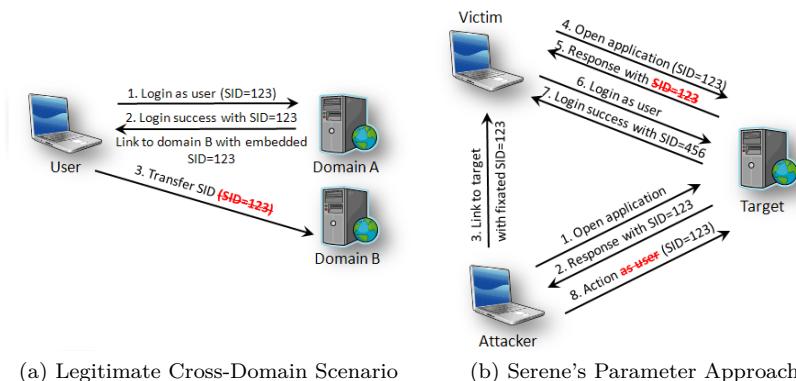


Figure 4.2: Schematic representation of the general approach mistakenly detecting a cross-domain parameter (left) and Serene's modified approach to allow such tokens to be sent once, but not return in the response (right)

The attack vectors described earlier are foiled by this approach. An externally injected link will contain a SID that is unknown to Serene, so it will be removed. Similarly, an attacker page embedding a SID in a URL to the target domain is not a logical scenario: a SID should be set by the domain that wants to maintain a session, not by another domain. Therefore, such SIDs are not stored in the database, so attaching it to a request will not be allowed by Serene. Additionally, SIDs injected by a script will not yet be present when Serene

examines the response, so they will not be stored in the database, therefore they will not be allowed on an outgoing request. Finally, if the attacker injects HTML code containing a fixated SID directly in the target application, it will collide with SIDs added by the application. Serene detects such collisions and either uses a previously stored SID, or removes all offending SIDs from the page.

#### 4.4.2 Parameter-Based Exchanges

Complex Web applications often share SIDs or SID-like tokens across domains using cross-domain requests with embedded parameters. Common examples are single sign-on services and applications that span multiple domains (e.g. `google.com` and `google.be`, etc.). Such a cross-domain interaction pattern shows similar characteristics as a parameter-based session fixation attack, and is also detected and prevented by Serene's general approach (Figure 4.2a).

From a security point of view, Serene's general approach is quite effective, since it stops potential session fixation attacks. From the usability point of view however, breaking functionality is to be avoided. Therefore, we slightly relax the general approach for SIDs embedded as a parameter: we allow them to be sent to the server only once, and make sure that they are not used to maintain a potentially fixated session. Practically, Serene remembers unknown SIDs embedded as a parameter when sending the request, and makes sure these unknown SIDs are removed from the corresponding response if present (e.g. in an element, script, header etc.). This relaxation (Figure 4.2b) allows a valid token to be exchanged, and prevents a fixated SID from doing harm. Even if a fixated SID is sent to the server, it can never be used to establish a fixated session, since Serene will prevent it to be used on subsequent requests.

#### 4.4.3 Session Identifier Identification

Serene's approach depends on the capability to distinguish session identifiers from other cookies and parameters. Earlier approaches to tackle this problem [185, 235] are not directly applicable for reasons we discuss in Section 4.6. Therefore, we used these approaches as inspiration for an elementary SID identification algorithm. Below, we discuss the way the algorithm works. In Section 4.5, we evaluate the algorithm on existing sites.

The first step in the algorithm is to check whether the key matches an extensive list of 45 known session identifier names. If not, we check whether the key/value pair passes all of the following three heuristics:

1. The key has to include an obvious part of a SID name (e.g. *sess* or *id*)
2. The value has to be sufficiently long (i.e. 10 or more characters).
3. The value has to be sufficiently random [185].

Each of the three heuristics serves a specific purpose. The first heuristic is required to rule out key/value pairs that contain a long, seemingly random value, but are in no way related to a SID. The second heuristic rules out short identifiers, such as product or article identifiers. The third heuristic rules out any non-random values, since they cannot serve as a valid session identifier anyway. Any key/value pair not matching these heuristics is either not session-related, or not fit to serve as a session identifier. For instance, a short or non-random value is easily brute forced.

#### 4.4.4 Prototype Implementation

The prototype of Serene is implemented as a Firefox add-on, available for any modern version of Firefox. Once Serene is installed, it protects against session fixation attacks without requiring any configuration.

Due to the extensive add-on support available within the Mozilla framework, the implementation of Serene is fairly straightforward. Cookie inspection and manipulation is done through the HTTP channel, which provides access to the request before it is sent out and to the response before it is processed. The HTTP channel is also used to detect outgoing SIDs embedded as parameters. Incoming parameters containing SIDs are extracted from the page contents and potentially harmful tokens are removed from the page before it is processed.

### 4.5 Evaluation

Serene's approach for both cookie-based and parameter-based session management successfully counters attack vectors *A1*, *A2*, *A3*, *A4* and a script-based *A6*. Self-reliant client-side protection against header-based attack vectors (*A5* and *A6*) is very challenging, since these attack vectors exhibit exactly the same behavioral patterns as the de facto session management techniques used in the majority of modern applications. Due to their damage potential, attacks using attack vector *A5* are scarce. In Section 4.6, we elaborate on attack vector *A6*.

In the remainder of this section, we evaluate two important aspects that determine the successful applicability of Serene: (i) does the SID identification

algorithm support a large majority of available sites, and (ii) is Serene compatible with available sites? From a study of Alexa’s top one million sites, we show that Serene already protects 83.43% of analyzed applications, a number that can be increased with future refinements. Additionally, we demonstrate that Serene fully preserves the functionality of 95.14% of sites.

### 4.5.1 Session Identifier Identification

To analyze the coverage of the SID identification algorithm proposed in Section 4.4.3, we collected the index pages of Alexa’s top one million sites, storing both the response headers and the response pages. We used `wget` to collect only the main page, without attempting to load any subresource (e.g. images, scripts, etc.), thus we only sent one request to each listed site. One exception is a response with a redirect, which we followed until it pointed at an actual page.

To assess the validity of the SID identification algorithm, we conducted a manual analysis of a subset of 1,000 sites. This analysis shows that the SID identification algorithm effectively filters out SIDs from other values. Out of 5,500 cookies, 1,953 are identified as SIDs. Of these 1,953 cookies, we only discovered 10 cookies that cannot be obviously classified as a SID.

Analyzing the set of top one million sites shows 472,834 sites ask the browser to set a cookie upon the first request. Running the SID identification algorithm on these cookies reveals that the cookies of 349,480 domains (73.98%) contain a session identifier: 266,305 domains use a SID with a known name and 98,305 use a SID that matches the three heuristics. Note that these numbers indicate that 15,130 domains use both a SID with a known name and a SID that matches the heuristics. Manual inspection of a subset shows that several sites use indeed multiple SID key/value pairs, for instance two different kind of identifiers (a session ID and a visitor ID) or different keys for the same SID value. Finally, analyzing the use of the *Domain* option on cookies containing a session identifier shows that 6.5% of 349,480 sites make the SID available for all subdomains.

These numbers suggest that of the 472,834 sites setting cookies, 123,354 do not include a session identifier in their cookies. We isolated these sites and conducted a follow-up study: similar to the first study, we fetched their index page twice, independently from each other. By comparing the cookie’s key/value pairs present in the response, we can detect potential false negatives in the SID identification algorithm: if the cookies of both responses are exactly the same, then these cookies cannot represent a session identifier, since a SID cannot be shared between two independent requests. The results show that of the 123,354 sites, 77,935 set at least one different cookie value on both requests. Applying the length and randomness heuristic suggests that 69,405 of these

domains actually set some kind of identifier. In total, this means that with the elementary SID identification algorithm, Serene already protects 349,480 domains out of 418,885 domains setting a SID, or 83.43%.

**Conclusion.** The analysis of the support of the elementary SID identification algorithm shows that Serene is able to protect a large majority of Alexa’s top one million sites. In Section 4.6 we elaborate on potential refinements of the SID identification algorithm, allowing us to increase the level of protection.

### 4.5.2 Application Compatibility

In the second part of the evaluation, we take a closer look at the impact of Serene’s protective measures on the functionality of available sites. We prepared a clean Firefox profile with Serene installed. We instructed Firefox to load each site using this clean profile, stopped Firefox after 25 seconds and collected statistics generated by Serene. Note that this process not only loads the index pages, but also all included resources, both within the domain and external, thus triggering Serene’s protective measures.

Our study shows that of the one million processed sites, Serene has no negative effect on the functionality of 524,014 (93.14%) of 562,538 sites that set cookies. A follow-up manual analysis of the most common impacted traffic patterns reveals that third party services, such as tracking, analytics or advertising, often trigger Serene’s protective measures. Several sites have even documented this behavior [30, 75]. Additionally, recent initiatives such as tracking protection lists [178] or Do-Not-Track [175] also aim at discouraging this behavior. Removing obvious instances of these services brings Serene’s compatibility to 95.55%.

**Conclusion.** The compatibility study of Serene’s impact on available Web applications shows that Serene fully preserves the functionality of 93.14% of sites. Not counting privacy-invasive third party services brings the level of compatibility up to 95.55%. For the remaining 4.45%, we suggest a follow-up user study to investigate the noticeable impact on an application’s functionality.

## 4.6 Discussion

**Refining SID Identification.** Earlier work already proposed an algorithm for client-side identification of session identifiers in cookies [185, 235]. Both

approaches aim at preventing session hijacking attacks, where the attacker steals the cookie containing a SID through JavaScript, allowing him to take over the session. The proposed client-side solution is to attach the *HttpOnly* option to an identified SID, which prevents the cookie from being read from JavaScript. SID detection happens using a selected list of known names, combined with heuristics on the value. Note that in case of false positives, the SID is prevented from being read in JavaScript, but is still sent to the server on outgoing requests.

Initially, we attempted to use such an algorithm for SID identification in Serene, but the algorithm produced quite a few false positives, both for cookies and for parameters. Unfortunately, a false positive in Serene means that the value will be removed from the request, so it will never be sent to the server. This effect is severely more disruptive than preventing JavaScript to access a cookie with probably a random SID. We addressed these problems with the elementary SID identification algorithm, as proposed in Section 4.4 and evaluated in Section 4.5.

In future work, we suggest to refine the elementary algorithm by carefully integrating the more generic, heuristic algorithms, in order to reduce the false negative rate. To support this suggestion, we ran the 77,935 domains that sent two different cookie values in two independent requests through SessionShield’s algorithm, which suggests that further refinement can extend support to 63,384 of these 77,935 domains, resulting in a total compatibility of 98.6%.

**Subdomain Attack Vector.** As mentioned before, Serene covers all session fixation attack vectors, except for header-based attacks (*A5* and *A6*). Attack vector *A6* is most likely to occur, and is launched through a **Set-Cookie** header that sets a cookie belonging to all subdomains. In order to launch such an attack, the attacker needs to control such a subdomain and needs to be able to set custom response headers (i.e. a **Set-Cookie** header).

Preventing these session fixation attacks at the client-side is currently not possible, because the pattern of an attack is very similar to a legitimate usage pattern, where a domain wants to set a SID belonging to all subdomains. Simply disallowing such a SIDs would break a substantial fraction of sites. Section 4.5 shows that already 6.5% (22,706 out of 349,480) of sites setting a SID on their index page use the *Domain* option. Bortz et al. [41] also state that existing applications depend on sharing cookies across subdomains.

## 4.7 Conclusion

In this chapter, we presented Serene, the first self-reliant client-side counter-measure against session fixation attacks, fully covering 4.5 out of 6 attack vectors. Unfortunately, complete client-side protection is very challenging, due to potential abuse of headers, the only legitimate mechanism currently available for Web applications. In a wide-scale study of Alexa's top one million sites, we have shown that Serene fully preserves 95.14% of functionality, while protecting 83.43% of investigated applications. Future refinement and a follow-up user study are the key to increase both the compatibility and coverage.

## Chapter 5

# Upgrading the HTTP Session Management Mechanism

The paper introduced in this chapter presents *SecSess*, a proposal to fundamentally address the security problems of bearer token-related session management mechanisms in the presence of *Web attackers* and *passive/active network attackers*. SecSess effectively prevents the *transfer of a session without authorization*, by introducing an integrity check to the HTTP requests, based on a shared secret. We have designed SecSess to be compatible with currently deployed middleboxes on the Web, such as Web caches or perimeter security devices, a feature that is lacking from related proposals. This paper is an extended version of the paper accepted at the *30th ACM Symposium on Applied Computing* (SAC 2015) [72].<sup>1</sup>

The main contribution of SecSess is a simple session management mechanism that addresses the fundamental threat of an unauthorized session transfer. SecSess achieves this security property by determining a shared secret between server and browser during the session establishment phase. The integrity of each request is validated using the shared secret associated with the established session. This effectively prevents the attacker from taking over the session, as he does not know the shared secret, or from fixating the session, as he cannot transfer his shared secret to the user's browser. SecSess is compatible with current deployment scenarios on the Web, which often use a mixture of HTTP and HTTPS channels, as well as a variety of middleboxes deployed throughout the network infrastructure.

---

<sup>1</sup>Philippe De Ryck is lead author of this paper.

Even though widespread TLS deployment remains the optimal deployment strategy, with which *SecSess* is fully compatible, the current state-of-practice shows that full TLS deployment across the Web may be an utopian dream. Therefore, we envision the upgrading of the HTTP session management mechanism within the current movement towards improving the security of the default plaintext channel, with techniques such as opportunistic encryption [188] being proposed to be included in the upcoming HTTP/2.0 [32] specification.

# SecSess: Keeping your Session Tucked Away in your Browser

**Abstract** Session management is a crucial component in every modern Web application. It links multiple requests and temporary stateful information together, enabling a rich and interactive user experience. Unfortunately, the de facto standard cookie-based session management mechanism is imperfect, which is why session management vulnerabilities rank second in the OWASP top 10 of Web application vulnerabilities [257]. While improved session management mechanisms have been proposed, none of them achieves compatibility with currently deployed applications or infrastructure components such as Web caches.

We propose SecSess, a lightweight session management mechanism that addresses common session management vulnerabilities by ensuring a session remains under control of the parties that established it. SecSess is fully interchangeable with the currently deployed cookie-based session management, and can be gradually deployed to clients and servers through an opt-in mechanism. Evaluation of our proof-of-concept implementation shows that SecSess introduces only a minimal performance and networking overhead. Furthermore, we empirically show that SecSess is effectively compatible with commonly used Web caches, in contrast to alternative approaches.

## 5.1 Introduction

Session management is a fundamental component of most modern Web applications, enabling the temporary storage of stateful information. The latter is crucial for widely-used features such as user authentication, authorization and transaction processing. As HTTP is stateless by design, this feature was added through the use of cookies [23] in a later phase. Unfortunately, cookies in the modern Web suffer from a number of imperfections, leaving cookie-based session management mechanisms and its users vulnerable. Since attacks on session management occur frequently and have a high impact – a successful compromise gives the attacker full control over the user’s session -, they are awarded a second place in the OWASP Top Ten of Web application security problems [257].

At the heart of a successful attack against session management lies an unauthorized session transfer. The most prominent example of such an unauthorized transfer is a session hijacking attack [185], where the attacker

steals a session identifier assigned to the user. A session hijacking attack can be carried out through different attack vectors, for example by injecting JavaScript code to exfiltrate the session identifier, or by eavesdropping on the network traffic, where the session cookie can be read from plaintext HTTP traffic. The enabler of such session transfer attacks is the use of the session identifier in current session management mechanisms. The session identifier acts as a *bearer token*, and the mere presence of this identifier in a cookie attached to the request suffices for legitimizing the request within the session.

Current best practices for preventing session transfer attacks advocate an HTTPS-only deployment combined with the *HttpOnly* and *Secure* cookie flags. Such a deployment only transmits the session cookie over an encrypted channel, and prevents the cookie from being accessed through JavaScript. While the benefits of HTTPS deployments are evident, wide scale adoption on the Web is impeded by several intricacies. One often cited issue is the performance impact, an argument that has lost most of its relevance on modern hardware [153]. Second, HTTPS deployments are disproportionately more complex compared to HTTP deployments, putting a significant burden on system administrators. Examples of such complexities are creating keys, monitoring and renewing certificates, dealing with browser-approved certificate authorities, preventing mixed-content warnings and deploying shared hosting using TLS's Server Name Indication extension [85], if supported by the client. Additional to the complexity of deploying HTTPS, a wide-scale transition to HTTPS severely obstructs the operation of the so-called middleboxes, machines in between the endpoints that cache, inspect or modify traffic. These middleboxes are essential parts of the Web infrastructure, for example by bringing the Web to developing nations through extensive caching and enabling efficient video transmission on mobile phone networks.

We acknowledge that wide-scale deployment of HTTPS remains imperative for securing the Web, but also recognize the long and tedious process. This explains why the recent revelations about pervasive monitoring on the Web have sparked multiple proposals looking to transparently upgrade the security properties of the HTTP channel when supported by the endpoints. One prominent proposal is to negotiate an encrypted HTTP channel without verifying the entities' authentication [189], which is even proposed as one of the available modes in the upcoming HTTP/2.0 specification. This eagerness to improve the security properties of the HTTP protocol, even by introducing them into the new version, shows that the HTTP protocol will be around for the near future. Therefore, it makes sense to not only upgrade the network-level protocol properties, but also take the opportunity to improve the security properties of session management on top of the HTTP protocol.

In this work, we propose *SecSess*, a lightweight session management mechanism

that effectively eradicates the bearer token properties of the session identifier in current cookie-based session management mechanisms. SecSess is fully interchangeable with the current cookie-based workflows, and can be enabled on an opt-in basis, supporting a gradual migration path. Additionally, SecSess incurs only a minimal computational and network overhead, carefully avoiding the introduction of additional requests and roundtrips. To our knowledge, SecSess is the only session management mechanism explicitly designed to be compatible with currently deployed Web infrastructure, such as the popular Web caches.

In summary, our contributions are:

- SecSess, a lightweight session management mechanism for use on both secure and insecure channels, guaranteeing that the session cannot be transferred without explicit authorization.
- a fully functional prototype implementation with the client-side component as a Firefox browser add-on, and the server-side component as a middleware for the Express framework on top of Node.js.
- a thorough evaluation of the prototype, showing that SecSess introduces little to no performance and networking overhead.
- empirical evidence of SecSess’s compatibility with Web caches, by browsing the Alexa top 1,000 sites with our prototype through the popular Squid and Apache Traffic Server transparent caching proxies.

The remainder of this chapter is organized as follows. Section 5.2 covers the state-of-practice of session management mechanisms, the current threats and objectives for new systems. In Section 5.3, we present SecSess, our session management mechanism that prevents unauthorized session transfers. We discuss our implementation and evaluation in Section 5.4, followed by an extensive discussion on deployment and compatibility with the current Web in Section 5.5. We conclude the chapter with related work (Section 5.6) and a brief conclusion (Section 5.7).

## 5.2 Background

Before introducing SecSess, we first explain cookie-based session management, the de facto standard for session management on the Web. Second, we define the relevant threat model for session management mechanisms, followed by the proposal of four design objectives we consider crucial for a new session management mechanism to be used over both secure and insecure channels.

### 5.2.1 Session Management on the Web

In a cookie-based session management mechanism, the server generates a random identifier for a session, and sends it to the browser using the *Set-Cookie* header. The browser stores the cookie in the so-called *cookie jar*, and whenever a request is sent to a domain for which cookies are present in the cookie jar, the browser attaches these cookies using the *Cookie* header, as illustrated in Figure 5.1.

Unfortunately, the bearer token properties of the session identifier make cookie-based session management vulnerable to unauthorized transferring of the session. One such attack is a *session hijacking* attack [185], where an adversary is able to steal a user's session identifier. Simply attaching this session identifier to crafted requests is typically sufficient to hijack the user's session, granting the adversary the same level of access as the user. Concrete attack vectors for a session hijacking attack are script-based cookie exfiltration using the *document.cookie* property, or eavesdropping attacks on the network, as aptly demonstrated by the Firesheep addon [48], which reduces a session hijacking attack to a point-and-click operation.

A second attack is *session fixation* [219], where the adversary forces the user's browser to use a compromised session. The aim of a session fixation attack is to have the user authenticate himself within a session known to the attacker, causing the server to store the user's authentication state in the attacker's session. A session fixation attack is typically carried out by writing to the *document.cookie* property.

Since these attacks are well-known and well-documented, several countermeasures are available. Most relevant are the *HttpOnly* and *Secure* cookie flags,

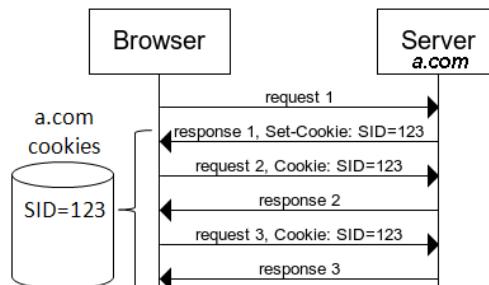


Figure 5.1: In cookie-based session management mechanisms, the server issues a cookie with a session identifier, which the browser stores in the domain-specific cookie jar and attaches to future requests.

which respectively restrict script-based access to cookies, and prevent the transmission of cookies over insecure channels. While these countermeasures offer adequate protection if deployed correctly, they do not fundamentally prevent unauthorized transfers, as the session identifier remains a bearer token. Additionally, these countermeasures are often not or incorrectly deployed, even within the Alexa top 100 sites [49], and new attacks that compromise secure deployments have been discovered [15].

### 5.2.2 Threat Model for Session Management

In an attack on the session management mechanism, the attacker aims to take control over the victim’s session. Based on this observation, we define the threat model for session management as a *session transfer attack*. Concretely, in a session transfer attack, the attacker is able to transfer a session defined between the victim’s browser and the target application to his own browser. Transferring the session grants the attacker the same privileges with the target application as the user holds.

We deliberately abstract the threat model to the conceptual level, as there are numerous concrete instantiations of a successful session transfer attack. One common example is performing a session hijacking or session fixation attack through attacker-controlled JavaScript. Another example are passive attacks on the network level, where an eavesdropping attacker can steal the session identifier from the network channel, either from the plaintext HTTP message or by performing traffic analysis attacks on an HTTPS channel [15]. In addition to passive network attacks, we also consider active network attacks to be in scope. In an active network attack, the attacker can manipulate, inject or drop packets on the network. Section 5.5.2 further discusses one particular case of the active network attack.

Next to these in-scope attack vectors of a session transfer attack, we consider attack vectors based on a compromise of the client-side or server-side infrastructure to be out of scope. The most common example are machines compromised by malware, both at the client and server side. Concretely, we expect an uncompromised machine and browser codebase at the client, as well as an uncompromised machine and Web application codebase at the server.

### 5.2.3 Objectives for New Session Management Mechanisms

Based on the discussion of the current cookie-based session management mechanism and its associated threats, we identify four design objectives for

a new session management mechanism. The first objective is a core design feature, focusing on preventing unauthorized session transfers. The three remaining objectives ensure the feasibility of deployment, covering induced overhead, compatibility with current applications and infrastructure, and a gradual migration path.

**Preventing Unauthorized Session Transfer.** State-of-practice session management mechanisms are vulnerable to session transfer attacks by design, due to their reliance on the session identifier as a bearer token. Newly designed session management mechanisms should actively try to prevent session transfer attacks. Additionally, session management mechanisms should still support authorized transfers, such as desktop-to-mobile synchronization at the client side, and load balancing at the server side.

**Minimal Overhead.** A crucial requirement for a new session management mechanism on the Web is a minimal overhead, well illustrated by browser vendors and Web application developers focusing on minimizing page load times. Overhead in the Web is twofold, with on one hand performance overhead, such as additional computations, and on the other hand networking overhead, with increased message sizes and additional roundtrips. Especially the latter is considered problematic, since it delays the processing of the page and the loading of sub-resources, such as style sheets, images, etc.

**Compatibility with Current Applications and Infrastructure.** A newly proposed session management mechanism should be compatible with the current Web and its peculiar deployment scenarios. Examples are the integration of third-party content in Web sites, and the redirection towards third-party service providers, such as a centralized authentication provider. On the infrastructure level, the Web deploys numerous middleboxes, such as Web caches at various levels and content inspection systems at network perimeters.

**Gradual Migration.** Finally, a new mechanism looking for adoption on the Web should support a gradual migration path, starting with early adopters on the client and server side, followed by a gradual increase of coverage in the Web. Key in this process is an application-agnostic opt-in session management mechanism, supporting implementation in current clients and server software or application development frameworks, thereby preventing the need for each individual application to incorporate the new mechanism. Additionally, backwards compatibility with parts of the Web that will not

quickly adopt the new mechanism is also important, since the Web cannot be updated in a single step.

## 5.3 SecSess

The essence of SecSess is establishing a shared secret used for session management between browser and server, which cannot be obtained by an attacker, thus effectively binding an established session to its initiating parties. In this section, we elaborate on our session management mechanism in two stages. First, we introduce the general idea and achieved properties without getting lost in details. In a second stage, we explain how these properties are achieved by highlighting each aspect of SecSess.

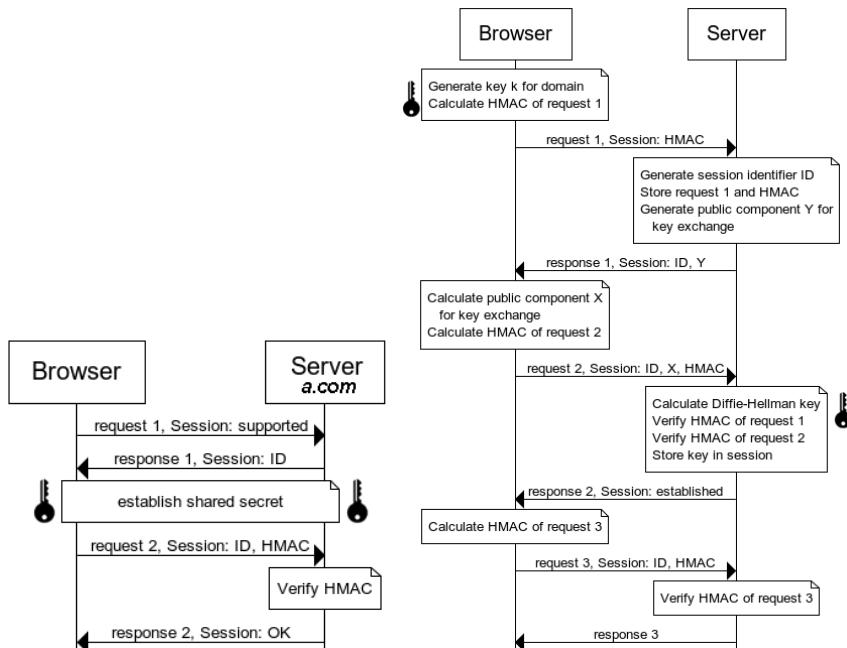


Figure 5.2: The flow of requests and responses used by SecSess, both as the *general idea* (a) and with *full details* (b).

### 5.3.1 General Idea

The general idea of SecSess is illustrated in Figure 5.2a. All session management instructions are contained in the newly introduced **Session** header, which keeps track of the parameters needed to establish a session, as well as the session identifier of the established session. Session management is based on a shared session secret, which is safely contained in the browser, inaccessible to any script code and never sent over the network. Using this shared session secret, we can generate a message authentication code (HMAC) for a request, which is sent to the server in the **Session** header. Using the shared secret associated with the session, the server can calculate the same HMAC in order to verify whether the received request is actually valid within the session.

Note that an eavesdropper can easily get hold of the session identifier, which is sent in the clear, but that the session identifier is only used to simplify bookkeeping. It is no longer the bearer token for the session, nor is it supposed to be secret. Using a simple incremental counter as an identifier is sufficient. An attacker attempting to use a stolen session identifier on a crafted request also needs the session secret to generate a valid HMAC for the request. Since this shared secret is safely contained within the browser, it cannot be obtained by an attacker.

### 5.3.2 Detailed Explanation

In this section, we further detail each aspect of SecSess in three steps: (i) the actual session management mechanism, (ii) establishing the shared session secret and (iii) the resulting request flow, which is identical to the flow in cookie-based session management mechanisms.

**Session Management.** Associating the server-side stored state with the appropriate requests is simplified by using a simple session identifier (ID) for bookkeeping. The session identifier is provided by the server using a **Session** response header (response 1 in Figure 5.2b). The browser attaches the session identifier to each request, using the newly introduced **Session** request header. Note that while the use of a session identifier strongly resembles traditional cookie-based session management, the session identifier is no longer considered to be a bearer token, and is useless without knowledge of the shared secret.

Instead of using the session identifier as the bearer token, SecSess uses the shared secret to add an hash-based message authentication code (HMAC) to the request, thereby legitimizing the request within the session. Since this HMAC

takes the request and the shared secret as input, only the browser and the server can compute the correct values. Incoming requests with an invalid HMAC are simply discarded by the server.

Note that the input for the HMAC should be chosen carefully. Technically, a network attacker can steal the valid HMAC from an eavesdropped request and attach it to a crafted request, having the crafted request reach the server first. In order to maintain a valid HMAC on the crafted request, the attacker can only modify the parts of the request that are not part of the input to the HMAC function. Including the URL of the request in this input prevents an attacker from directing the request to a different destination, but still allows him to modify sensitive information in the request headers and body (e.g. the destination account of a wire transfer). Therefore, the HMAC also covers the request headers containing sensitive data,<sup>2</sup> and, if present, the request body. Covering the URL, request headers and request body in the HMAC does not prevent an attacker from taking the valid HMAC value and attaching it to a crafted request. However, it does ensure that the attacker cannot change the sensitive data, hence limiting the contents of the crafted request to those of the original request, thereby reducing the problem to the common *double submission* problem [236].

**Establishing the Shared Secret.** The shared session secret, needed to compute and verify HMACs on requests, is established using the Hughes variant [128, 217] of the Diffie-Hellman key exchange algorithm (Figure 5.3), which allows to exchange the key even in the presence of eavesdropping attackers. In Figure 5.2b, the server sends his public value ( $Y$ ) after seeing the first request, in which the browser indicates support for the `Session` header. Using the server's public component  $Y$ , the browser can calculate the second public part ( $X$ ), which the server needs to calculate the key. In the next request, the browser sends the public value  $X$ , allowing the server to calculate the full key and verify this and any subsequent requests, effectively establishing the session, as acknowledged in the second response.

Note that the advantage of the Hughes variant of Diffie-Hellman is that the browser can compute the key before the first request is sent. This is required to attach an HMAC to the first request, so the server can verify that the sender of the first and second request are in fact the same. Omission of the first HMAC allows an eavesdropper to respond to the first response, injecting his key

---

<sup>2</sup>Concretely, we include the following standard HTTP headers: *Authorization*, *Cookie*, *Content-Length*, *Content-MD5*, *Content-Type*, *Date*, *Expect*, *From*, *Host*, *If-Match*, *Max-Forwards*, *Origin*.

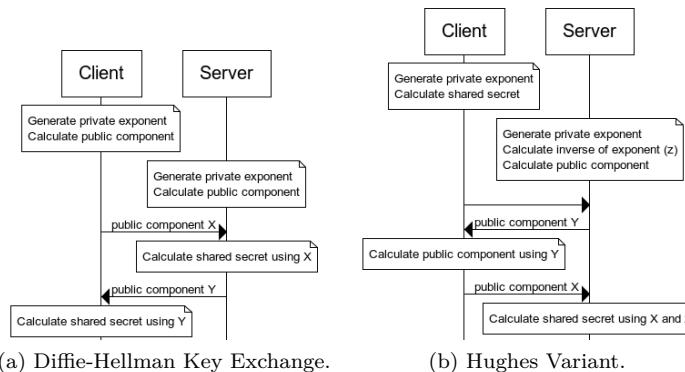


Figure 5.3: The main advantage of the Hughes variant of the Diffie-Hellman key exchange is that the client already knows the secret at the start of the exchange, enabling its use in the first request. SecSess uses this property to add an HMAC to the first request.

material into the session, which is problematic when the first request already caused some server-side state to be stored in the session.

**Preserving the Request Flow.** By design, SecSess is an application-agnostic session management mechanism, preserving the same flow of requests and responses as a currently deployed cookie-based session management mechanism (Figure 5.4). This property supports a gradual deployment, where client and server software can be upgraded to opt-in to SecSess next to cookie-based session management. If the client does not support SecSess, no **Session** header is sent, so the server simply defaults to cookie-based session management. Alternatively, if the client supports SecSess, but the server does not, the **Session** header will be ignored by the server, and the default cookie-based session management mechanism will be used.

### 5.3.3 Handling Modified Request Flows

Since the Web is a complex distributed system, where multiple simultaneous requests are fired by the browser, request flows often differ from the flows drawn on paper. One example of a modified flow are requests that arrive at the server in a different order than they were sent. A second example are middleboxes changing the request flow, such as a Web cache responding to a request, which

will thus not be sent to the server. Since these scenarios are common in the Web, it is important that they are robustly handled by a newly introduced session management mechanism.

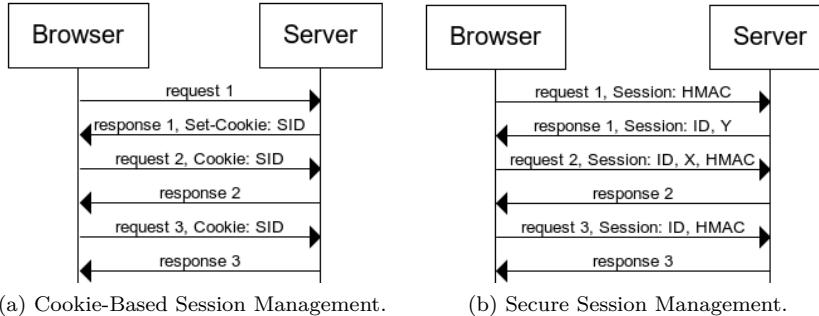


Figure 5.4: The traffic flow of SecSess is fully compatible with the traffic flow of existing cookie-based session management mechanisms, allowing a smooth transition between both mechanisms.

The design of SecSess explicitly takes modified request flows into account, and effectively achieves full compatibility with currently deployed Web caches, both within the browser and on the intermediate network. First, by only adding integrity protection, the caching of content is effectively enabled. Second, SecSess is robust enough to deal with out-of-order requests and cached responses, which is fairly trivial once a session is established, but challenging during establishment. If the client's public component (request 2 in Figure 5.2b) would get lost in transit, for example when an intermediate cache responds to a request, the server would not be able to complete the session establishment, effectively breaking the protocol. Concretely, SecSess addresses this by continuing to send the public component as long as the server has not confirmed the session establishment (response 2 in Figure 5.2b), effectively preventing it from getting lost in a modified request flow. We discuss the concrete impact of this decision during the performance evaluation.

## 5.4 Implementation and Evaluation

To show the feasibility of SecSess on the Web, as well as to support evaluation, we created a proof-of-concept implementation. At the client side, we have extended the Firefox browser with support for SecSess, heavily leveraging the support of OpenSSL's crypto library. At the server side, we have implemented

a session management middleware module for the Express framework, which runs on top of Node.js, an event-driven bare metal Web server. The middleware amounts to a mere 113 meaningful lines of code, and a binary module linking the OpenSSL library is 178 meaningful lines of code.

The prototype implementation of SecSess enables a thorough evaluation, both on performance as on compatibility with the current Web, as covered in the remainder of this section. The next section discusses both client and server-side strategies for evolving the prototype into a practically deployable solution.

### 5.4.1 Security

The security evaluation of SecSess with regard to the proposed in-scope threat model considers several concrete attack vectors. A first is the capability to run attacker-controlled scripts within the context of the target application. A session hijacking or session fixation attack using this attack vector will no longer succeed, since none of SecSess's data is available to the JavaScript environment. Additionally, the `Session` request and response headers contain only public information, of no use to an attacker.

Session transfer attacks can also be performed on the network level. Eavesdropping attacks on the session management mechanism are effectively mitigated by SecSess, since the shared secret used for calculation of the HMACs is never communicated over the wire, and the Hughes variant of the Diffie-Hellman key exchange can withstand passive attacks. Next to passive attacks, an attacker can also try to modify existing requests, or re-attach a valid HMAC to a crafted request. Such attempts will fail as well, because the HMAC is based on the contents of the request, effectively preventing any modifications to go unnoticed.

### 5.4.2 Overhead

**Performance Overhead.** Figure 5.5 shows the performance overhead induced by SecSess on a session establishment timeline. To get correct measurements, we calculated 100 data points for each step, which contain the average computation time of 100 runs each, executed from within JavaScript code, both on the client-side (browser add-on) as the server side (Node.js).<sup>3</sup>

---

<sup>3</sup>Experiments have been performed in a VirtualBox VM (Linux Mint 15), which was assigned 1 Intel i7-3770 core and 512 Mb of memory.



Figure 5.5: SecSess adds an average 4.3 milliseconds to the session establishment. The first step, where the shared secret is generated at the client side and the computation parameters are generated at the server side, takes a bit longer, but can be pre-computed offline or during idle times.

Most notable results are the very limited overhead at the client-side, especially after the session has been established (from request 3 onwards). At the server side, there is a significant pre-calculation overhead (212ms) for generating the required parameters. This overhead is induced by the Hughes variant of the Diffie-Hellman key exchange, which requires the inverse of the server's private component. Note that these parameters are session-independent, and can be pre-calculated offline in bulk, and read from a file on a per-need basis. After the parameters have been calculated, the additional overhead for actually establishing and maintaining a session is negligible.

**Network Overhead.** In a Web context, network overhead can be caused by increased message sizes, but also by introducing additional requests or round trips in the flow of requests. By design, SecSess follows the same sequence of requests and responses used in currently existing applications, which deploy cookie-based session management mechanisms, so no additional requests or round trips are required.

For brevity reasons, we do not go into detail about the exact header sizes as shown in Figure 5.6. However, based on the average header sizes in the cookie-based session management mechanisms of the Alexa top 5,000 sites, SecSess leads to 25.58% reduction in the size of the session management header of requests. On the contrary, SecSess leads to increased header sizes during the session establishment phase, which ends after the server has confirmed the session establishment (response 2 in Figure 5.2b). These header sizes increase due to the transmission of the public components of client and server. Concretely, the request headers during session establishment suffer a 867.44% increase, and the response header 9.19% increase.

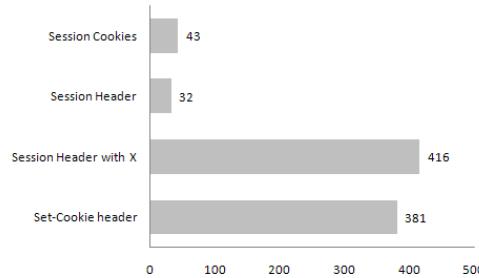


Figure 5.6: The most frequent **Session** header is 32 bytes, which is less than the average session cookie for the Alexa top 5,000 sites. During session establishment, SecSess induces slightly larger headers, which are still within ranges common on the Web, as indicated by the average *Set-Cookie* header size for the Alexa top 5,000.

### 5.4.3 Compatibility with Web Caches

Web caches are widely deployed throughout the Web, enabling faster page loads and limiting the required bandwidth. Caches are often deployed in a transparent way, where they intercept HTTP traffic, and respond when they have the resource in cache. When a cache responds to a request, the request is never forwarded to the target server, resulting in a modified request flow. As stated in one of the proposed design objectives, newly proposed session management mechanisms should be able to cope with infrastructure components of the Web modifying request flows.

SecSess is robustly designed to be compatible with such modified request flows. We have confirmed this compatibility empirically by running experiments with two popular caches, Squid [225] and Apache Traffic Server [238], configured as a forwarding proxy. In our setup (Figure 5.7a), we add SecSess session management on top of the requests sent between the browser and the Web servers of the Alexa top 1,000 sites. Since these servers do not know about SecSess, we have added a dedicated SecSess-proxy in between, which will handle the SecSess session management with the browser (full arrows), while forwarding the request to the actual Web server (dashed arrows). Finally, we add the cache in between the browser and the SecSess-proxy. This setup allows us to test the establishment and maintaining of a session with traffic patterns from the Alexa top 1,000 sites. Additionally, when the cache responds to a request, the SecSess-proxy will never see the request. This effectively allows us to verify the robustness of SecSess when dealing with modified request flows. The results are

shown in Figure 5.7b.

To maximize the potential of the cache, we visited each site in the Alexa top 1,000 twice. For the Squid run, 52,947 requests were sent to 5,167 distinct hosts. Of these requests, 5,008 were cached, of which 830 during session establishment, and 4,178 when an established session was already present. For the Apache Traffic Server run, we observed 44,173 requests to 4,660 hosts in total, of which 4,263 were served from the cache. 1,169 cached responses occurred during session establishment, and 3,094 with an established session. During these requests, SecSess robustly handled session management, without losing an established session, or failing to establish a session.

## 5.5 Discussion

In this section, we discuss practical deployment strategies for rolling out SecSess in the current Web. Additionally, we also elaborate on session-related attacks, and why they fall beyond the realm of the session management mechanism.

### 5.5.1 Deploying SecSess

Implementing SecSess’s functionality at the client side is best done within the browser, who is responsible for managing sessions, sending requests and receiving responses. A browser-based implementation has full control over all outgoing requests, regardless whether they originate from within a page, JavaScript or the browser core. Currently, we implemented SecSess as a lightweight add-on, enabling easy development of a proof-of-concept for evaluation with the latest

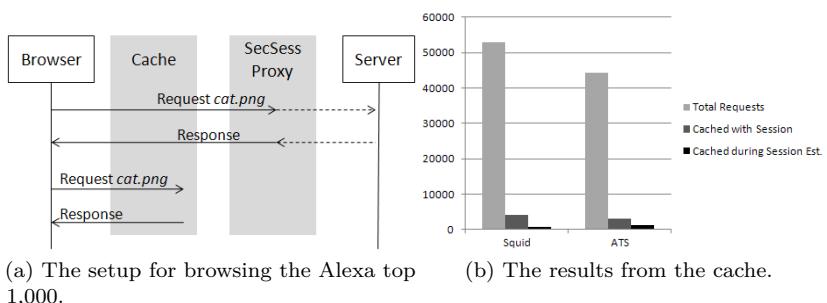


Figure 5.7: The setup and results of the cache compatibility experiment.

browser. The long-term deployment strategy integrates the current codebase into the browser core, enabling SecSess as an opt-in mechanism at the client side.

At the server side, SecSess can be supported through two complementary implementation strategies. The first option is to integrate SecSess directly into the Web platform, while remaining transparent to the target applications, for example as a module within the Web server (e.g. as an Apache module), or as part of the underlying application framework (e.g. our Express middleware implementation). The alternative strategy focuses on supporting legacy applications, where the underlying frameworks cannot easily be upgraded. Implementing SecSess as a reverse proxy in front of the server provides secure session management towards the browser while shielding the server from these changes.

Since SecSess follows the same traffic flow as current cookie-based session management mechanisms, it remains fully compatible with advanced Web technologies. One common example are load balancing techniques deployed at the server side, where multiple servers serve requests for the same target application. Since cookie-based mechanisms already require session tracking or state sharing between servers, SecSess can benefit from the same technology.

### 5.5.2 Related Attacks

A session management mechanism is responsible for establishing and maintaining a session, threatened by adversaries aiming to gain control of a session established between a user's browser and the target application. SecSess effectively prevents these *session transfer attacks*, while meeting the objectives proposed earlier. In this section, we discuss two related attacks which are out of scope for a session management mechanism, but are nonetheless related.

**Replay Attacks.** In a replay attack, the attacker replays earlier-recorded traffic towards the target application, making the application believe the user performed the same operations twice. Common protection mechanisms against replay attacks are the use of nonces or counters, which in turn require storing previously seen nonces, or a sliding window of valid counters, as often applied in various network protocols. While preventing replay attacks at the application level is out of scope for a session management mechanism, SecSess does ensure that the properties of an HTTP request cannot be modified. Essentially, an attacker can only replay an exact copy of an HTTP request, which reduces the problem to a user hitting the refresh button in his browser, or impatiently

submitting a form twice. Therefore, we have deliberately decided to exclude replay protection from SecSess, since these *double submission* issues [236] have to be handled within the application anyway.

**Man-in-the-Middle Attacks.** A special case of an active network attack is a man-in-the-middle attack, where an attacker actively manipulates the network infrastructure, positioning himself in between the user’s browser and the target application. If this occurs before a session has been established, the victim’s browser will establish a session with the attacker, after which the attacker establishes a session with the target application. Note that this does not violate SecSess’s security guarantees, since the session is still tied to its initiating parties, albeit that one of the parties is the attacker. Protecting against man-in-the-middle attacks requires explicit entity authentication, as offered by the TLS protocol. Integrating such protection in a session management mechanism would result in a “re-invention” of the TLS protocol on a higher level, an effort that does not seem useful. Note however that SecSess is only vulnerable to man-in-the-middle attacks during session establishment, but is resilient against these attacks after the shared secret has been exchanged.

## 5.6 Related Work

Related work offers several proposals to tackle the current session management problems. While these approaches offer significant benefits over cookie-based mechanisms, they only partially meet the objectives defined in Section 5.2.3. Additionally, many of the proposals depend on the presence of a TLS-channel, and do not withstand passive network attacks when such a channel is unavailable.

**SessionLock.** SessionLock [5] uses a JavaScript library to augment requests with an HMAC based on a shared session secret. The session secret is established over a TLS channel and stored in a secure cookie. For HTTP pages, it is stored in the fragment identifier, a part of the URL that is never sent over the network. SessionLock also supports a non-TLS scenario, where the client performs an out-of-band Diffie-Hellman key exchange with the server.

The idea behind SessionLock is similar to the idea behind SecSess, but the implementation differs significantly. The implementation as a JavaScript library not only fails to protect the session against script-based attacks, but also requires significant changes to existing applications, as all requests are through AJAX calls.

**BetterAuth.** BetterAuth [142] is an authentication protocol for Web applications, offering protection against several attacks, including network attacks, phishing and cross-site request forgery. BetterAuth considers a user’s password to be a shared secret, and uses that shared secret to agree on a session secret over an insecure channel. The session secret is used to sign requests, offering authenticity.

BetterAuth offers strong security properties, and is even capable of protecting against man-in-the-middle attacks. However, BetterAuth requires the exchange of the password over a secure channel, as well as the modification of existing applications. Additionally, BetterAuth depends on the password, it is incompatible with current third-party authentication services.

**HTTP Integrity Header.** The HTTP Integrity Header [112] is an expired draft proposing to add integrity protection to HTTP, which includes a session management mechanism. The header depends on a key exchange, either over TLS or with a traditional Diffie-Hellman exchange, after which the integrity of the selected parts of a message is protected.

The HTTP Integrity header actually shares the same idea as SecSess, using a shared secret for session management and integrity properties. However, the HTTP Integrity header uses the original Diffie-Hellman protocol, which only establishes a secret at the client after the first request and response have been exchanged. This leaves the setup phase of the session vulnerable to passive network attacks. Additionally, the HTTP Integrity header does not account for the adverse effects of caches or out-of-order requests during session establishment.

**One-Time Cookies.** One-Time Cookies [59] proposes to replace the static session identifier with disposable tokens per request, similar to the concept of Kerberos service tickets. Each token can only be used once, but using an initially shared secret, every token can be separately verified and tied to an existing session. To share the initial credential, One-Time Cookies depends on the use of TLS during the authentication phase.

One-Time Cookies would be a good replacement for traditional cookie-based session management mechanisms. However, since the initialization must be done over TLS, it loses its security properties when deployed for applications that only use HTTP, making a short-term deployment infeasible.

**TLS Origin-Bound Certificates.** Origin-Bound Certificates (OBC) [81] is an extension for TLS, that establishes a strong authentication channel between browser and server, without falling prey to active network attacks. Within this secure channel, TLS-OBC supports the binding of cookies and third-party authentication tokens, which prevents the stealing of such bearer tokens.

TLS-OBC offers strong security guarantees, and is able to eliminate the bearer token-properties of sensitive cookies. However, since TLS-OBC obviously depends on a TLS-only deployment, it is not a feasible solution for securing current and future HTTP deployments.

## 5.7 Conclusion

The current de facto standard for session management on the Web is extremely vulnerable to session transfer attacks. Best practices advocate deployment over TLS with the appropriate cookie flags, but several factors hinder such wide-scale TLS deployment. Alternative proposals for new session management mechanisms either also depend on the presence of a TLS channel, or fail to robustly deal with the Web’s infrastructure, such as widely deployed Web caches.

We have proposed SecSess, a lightweight session management mechanism that prevents unauthorized session transfers, and is explicitly designed to be compatible with the current Web. SecSess preserves the flow of requests observed today with cookie-based session management mechanisms, hence SecSess is compatible both with current infrastructure as with current Web applications. We empirically illustrated this compatibility by visiting the Alexa top 1,000 sites through two popular caching proxies, observing no dropped sessions, during or after establishment.



# Chapter 6

## Client-Side Detection of Tabnabbing Attacks

The paper covered in this chapter introduces *TabShots*, the first effective client-side countermeasure against tabnabbing attacks. Tabnabbing, a surprisingly sly variant of phishing, allows a *Web attacker* to obtain the user’s credentials, thereby enabling the threat to *impersonate a user by establishing a session*. Since the user’s credentials allow the attacker not only to establish a session, but also to bypass any additional in-session authentication checks, a phishing or tabnabbing attack can lead to worse consequences than any of the previously discussed attacks. This paper was presented at the *8th ACM Symposium on Information, Computer and Communications Security* (ASIACCS 2013) [73].<sup>1</sup>

As will become clear throughout this chapter, in a tabnabbing attack, an innocuous page visually disguises itself as a login page for a legitimate application when the user is not paying attention. Since the tabnabbing page is fully controlled by the *Web attacker*, detection mechanisms that depend on the underlying structure of the page, such as the HTML elements and attributes, are likely susceptible to evasion. Therefore, *TabShots* is based on screenshots of a browser’s tab, essentially analyzing the same view as seen by the user. To prevent the user from entering credentials in a fraudulent phishing form, *TabShots* highlights the parts of the page that have changed since the last visit, visually alerting the user of a “phishy” situation.

The approach and proof-of-concept implementation of *TabShots* show the

---

<sup>1</sup>Philippe De Ryck and Nick Nikiforakis took the lead on this paper. Philippe focused on implementation and evaluation.

potential of visual client-side mitigation techniques against phishing-based attacks. Even though alternative authentication systems are being proposed on a regular basis, all promising to eradicate the traditional credentials, username/password-based authentication is likely to stick around for a while. Alternatives to a visually supported detection mechanism could be browser-generated warnings when entering a username in an unknown form, or the use of secure password managers, which would require explicit configuration on unknown Web pages. Unfortunately, users are often the weakest link in the chain, causing a significant amount of attacks to be targeted at them.

In hindsight, our work on TabShots highlights an interesting, alternative approach towards client-side mitigation techniques, especially compared to CsFire and Serene, which modify the browser's security policies. Visually comparing screenshots of tabs posed alternative challenges, such as performing image processing tasks from within JavaScript, and making sure the highlighting of the changed parts occurs fast enough so users can be warned in time.

# TabShots: Client-Side Detection of Tabnabbing Attacks

**Abstract** As the Web grows larger and larger and as the browser becomes the vehicle-of-choice for delivering many applications of daily use, the security and privacy of Web users is under constant attack. Phishing is as prevalent as ever, with anti-phishing communities reporting thousands of new phishing campaigns each month. In 2010, tabnabbing, a variation of phishing, was introduced. In a tabnabbing attack, an innocuous-looking page, opened in a browser tab, disguises itself as the login page of a popular Web application, when the user’s focus is on a different tab. The attack exploits the trust of users for already opened pages and the user habit of long-lived browser tabs.

To combat this recent attack, we propose TabShots. TabShots is a browser add-on that helps browsers and users to remember what each tab looked like, before the user changed tabs. Our system compares the appearance of each tab and highlights the parts that were changed, allowing the user to distinguish between legitimate changes and malicious masquerading. Using an experimental evaluation on the most popular sites of the Internet, we show that TabShots has no impact on 78% of these sites, and very little on another 19%. Thereby, TabShots effectively protects users against tabnabbing attacks without affecting their browsing habits and without breaking legitimate popular sites.

## 6.1 Introduction

Phishing, the process that involves an attacker tricking users into willingly surrendering their credentials, is as prevalent as ever. PhishTank, a volunteer-driven site for tracking phishing pages [190], in their latest publicly available report, reported a total of 22,851 valid phishing attempts just for July of 2012. In these attacks, an attacker targets the user and capitalizes on a user’s inability of distinguishing a legitimate page from one that looks legitimate but is actually fraudulent. Phishing attacks can be conducted both on large and small scale, depending on an attacker’s objectives. The latest publicized attack against the White House, involved the use of “spear phishing”, a type of phishing that is targeting highly specific individuals and companies [160].

In 2010, Aza Raskin presented a new type of phishing attack which he called “tabnabbing” [201]. In tabnabbing, the user is lured into visiting a malicious

site, which however looks innocuous. If a user keeps the attacker's site open and uses another tab of her browser to browse to a different Website, the tabnabbing page takes advantage of the user's lack of focus (accessible through JavaScript as `window.onBlur`) to change its appearance (page title, favicon and page content) to look identical to the login screen of a popular site. According to Raskin, when a user returns back to the open tab, she has no reason to re-inspect the URL of the site rendered in it, since she already did that in the past. This type of phishing separates the visit of a site from the actual phishing attack and could, in theory, even trick users who would not fall victim to traditional phishing attacks.

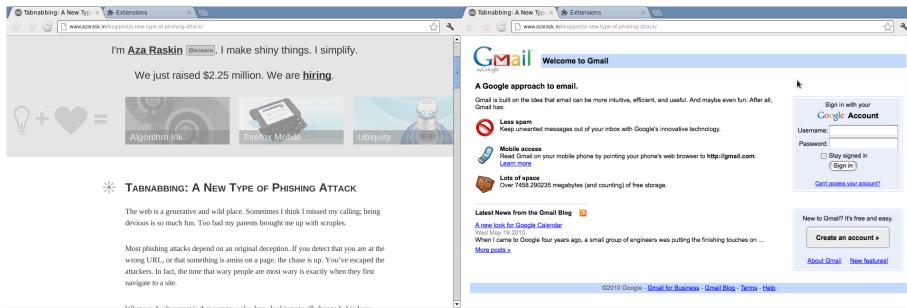


Figure 6.1: A seemingly innocuous page on the left performs a tabnabbing attack once the user switches focus, resulting in the page on the right [201].

In this chapter we present TabShots, a countermeasure for detecting changes to a site when its tab is out of focus. TabShots allows a browser to “remember” what the tab looked like before it lost focus, and compare it with the appearance after regaining focus. More precisely, whenever a tab is fully loaded, TabShots records the favicon<sup>2</sup> and captures a screenshot of the visible tab. Whenever a user revisits a tab, a new capture is taken and compared to the previously stored one. If any changes are detected, the user is warned by adding a visual overlay on the current tab, showing exactly the content that was changed, assisting the user in distinguishing between legitimate changes and tabnabbing attacks. Our system is based on the user’s visual perception of a site and not the HTML representation of it, allowing TabShots to withstand attacks that straightforwardly circumvent previously proposed, tabnabbing-detection systems. We implement TabShots as a Chrome add-on and evaluate it against the top 1,000 Alexa sites, showing that 78% of sites fall within a safe threshold of less than 5% changes, and an additional 19% fall within the threshold of less

<sup>2</sup>The small icon displayed in the tab’s title space

than 40% of changes. This means that TabShots effectively protects against tabnabbing attacks, without hindering a user’s day-to-day browsing habits.

The rest of this chapter is structured as follows: In Section 6.2 we first explore the original tabnabbing attack and then discuss possible variations taking advantages of the different implementations of the tabbing mechanism in popular browsers. In Section 6.3 we describe in detail the workings of TabShots and our implementation choices. In Section 6.4 we evaluate TabShots on security, performance and compatibility against the Alexa top 1,000. In Section 6.5, we briefly describe how TabShots could be deployed on the server-side to create tabnabbing blacklists and expand protection to all users. In Section 6.6 we discuss the related work and conclude in Section 6.7.

## 6.2 Background

### 6.2.1 Anatomy of a Tabnabbing Attack

Tabnabbing relies on the tab mechanism, which is common in all modern browsers. Users visit Websites, but instead of navigating away from that Website when they want to consume the content of a different Website, they open a new tab, and use that tab instead. The old site remains open in the old tab, and many tabs can accumulate over time in a user’s browser. A 2009 study of user’s browsing habits revealed that users have an average of 3.2 tabs open in their browsers [84]. We expect that today, this number has increased, due to the sustained popularity of social networking sites and Web applications that constantly update a user’s page with new information. The latest features introduced by browsers attest to this popularity of multiple open tabs, since they give the user the ability to “pin” any given tab to the browser and treat it as a Web application.

The steps of a tabnabbing attack as presented by Raskin [201] are the following:

1. An attacker convinces the user to visit a Website under his control. This Website appears to be an innocuous site that is not trying to fool the user into giving up her credentials. What the attacker must do, is convince the user to keep this tab open, and browse to a different Website. This is easily achieved in a wide range of ways, for instance by providing an article that is both very interesting, but also too long to read in a single go, or some sort of free product that will be available in the near future. Directing the user away from the attacker’s site is straightforward by

- adding the `target="_blank"` attribute to interesting hyperlinks, so that new links automatically open in a new tab or window.
2. JavaScript code running in the attacker's Website is triggered when the current window has lost focus, by registering to the `window.onBlur` event handler.
  3. The user keeps the attacker's Website open and uses other tabs to surf the Internet.
  4. The attacker realizes that his window is currently not in focus, and, after a possible delay of a few seconds in order to make sure that the user is busy consuming other content, changes the title, favicon and layout of the page to mimic the login screen of a Web application, for instance the user's Web mail or social networking site. The attacker can choose a default Web application (like Gmail) under the assumption that most users have a Gmail account or can combine the tabnabbing attack with a history-revealing attack [139, 253], and present the login of a Web application that he knows is visited in the past by the user. This process is also shown in Figure 6.1
  5. At some point in the future, the user recognizes a tab with a familiar favicon (e.g. GMail) and unwittingly opens the attacker-controlled tab. At this point, the user is no longer checking the URL of the Website, since it is a Website that she opened in the past and thus "trusted". Given a convincing login screen, the user proceeds into typing her credentials in the given forms which are then transferred to the attacker, thus completing the tabnabbing attack.

The main difference between tabnabbing and traditional phishing attacks is that the fake login form is decoupled from the visit of the malicious Website. Thus, users who have been trained to spot phishing attacks by immediately checking the URL of the page they open, may fall victim to this variant of phishing. This "delayed maliciousness" can also be used to evade detection by any automated honeyclients which may be autonomously searching for phishing pages based on various heuristics [255]. If the honeyclient does not stay for long enough on the malicious page, or does not trigger the `window.onBlur` event, then the actual phishing page will never be shown and the attacker can avoid detection.

### 6.2.2 Overly Specific Detection

In the previous section, we described the anatomy of a tabnabbing attack, exactly as it was first presented by Raskin in 2010 [201]. According to Raskin,

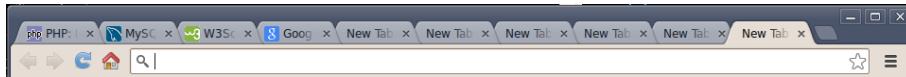


Figure 6.2: Chrome keeps all tabs visible but shrinks the space allotted to each tab

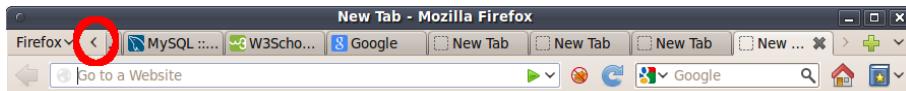


Figure 6.3: After a number of tabs, Firefox hides older tabs in order to make space for the new ones

an attacker needs to change three things in order to conduct a successful tabnabbing attack: the page's title, the page's favicon and the page itself. Accordingly, currently known countermeasures depend on changes in these three properties, or include even more specific tabnabbing characteristics (more details in Section 6.6). This overly specific detection gives the attacker more flexibility to avoid detection.

One example of such flexibility is carrying out a tabnabbing attack without changing the title of the tab, simply by taking advantage of the tabbing behavior within a browser. While conducting our research, we noticed that different browsers behave differently when a user has many open tabs in one window. Figure 6.2 and Figure 6.3 show how Chrome and Firefox handle many open tabs. Chrome, starts resizing the label of each tab, in an effort to keep all tabs visible. Here, one can notice that most of the title of each tab is hidden while favicons remain visible. On the other hand, Firefox starts hiding tabs which the user can access by clicking on the left arrow (circled in Figure 6.3). Moreover, Firefox preserves the title bar above the tabs, which Chrome dispenses in an effort to maximize the amount of space available for HTML.

In the case of Chrome, assuming that a user has many tabs open, the attacker can avoid the title change altogether, since it will likely not be visible to the user anyway.

In the next section, we present TabShots, which detects tabnabbing attacks using visual comparison. Since TabShots does not depend on fine-grained detection properties, we leave no room for an attacker to sneak through.

## 6.3 TabShots Prototype

### 6.3.1 Core Idea

As discussed before, a successful tabnabbing attack depends on the user visiting a malicious page, shifting focus to a different tab and returning at some point, after which the malicious page has changed its looks to resemble a popular application's login form. In itself, a tabnabbing attack is extremely obvious to detect, since a convincing phishing page will differ from the previous content. Detection is however complicated by the tab being out of focus, and the user placing some trust in previously opened and visited tabs.

TabShots takes advantage of these obvious changes needed by a successful tabnabbing attack, by remembering what a tab looks like before it loses focus, and comparing that to what it looks like when it regains focus. Any changes that happened in the background will be detected, and communicated to the user by means of a colored overlay. This allows the user to decide for herself whether the changes are innocent (e.g. an incoming chat message) or malicious masquerading (e.g. a login form and GMail logo popping up). Figure 6.4 shows how TabShots detects the tabnabbing attack from Figure 6.1. This non-intrusive behavior guarantees compatibility with all existing sites, since changes are only highlighted and not blocked or prevented.

Our approach is purely built on the visible content of a tab, exactly as the user perceives it. This yields several advantages compared to techniques analyzing the structure and contents of a page. TabShots is invulnerable to HTML, CSS or JavaScript trickery, aimed at circumventing tabnabbing countermeasures (see Section 6.6), scrolling attacks or other obfuscation attacks.

### 6.3.2 Implementation Details

TabShots is currently implemented as an add-on for Google Chrome,<sup>3</sup> but could easily be ported to other browsers supporting an add-on system, provided they offer a reliable way to capture screenshots of tabs.

In the following paragraphs, we discuss several implementation techniques and strategies for the major components of TabShots.

---

<sup>3</sup>A prototype of TabShots is available at <http://people.cs.kuleuven.be/~philippe.deryck/papers/asiaccs2013/>

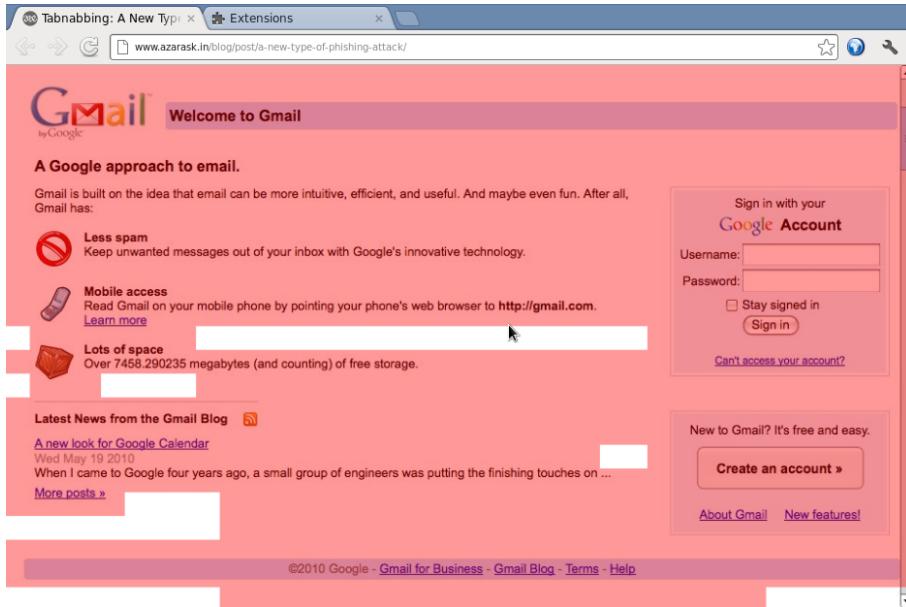


Figure 6.4: The overlay generated by TabShots for the attack from Figure 6.1. Here, only certain parts of the white background remained unchanged.

**Capturing Tabs.** TabShots records the favicon and captures screenshots of the currently focused tab at regular intervals, keeping track of the latest version. This latest snapshot will be the basis for comparison when a tab regains focus. Capturing a screenshot of a tab in Google Chrome is trivial, since the browser offers an API call to capture the currently visible tab of a window. Capture data is stored as a data URL [173].

Capturing snapshots of a tab at regular intervals is a deliberate design decision, allowing TabShots to handle changes that happen in a tab while it is in focus. These changes typically occur in highly dynamic applications, such as Facebook or GMail, which often use AJAX techniques to dynamically update the contents of their pages. Ideally, a tab could be captured right before it loses focus, but since Google Chrome does not offer such an event, this feature cannot be implemented without a severe usability and performance penalty.

**Comparing Tab Snapshots.** When a tab regains focus, TabShots needs to compare the current snapshot data with the stored data and detect any differences. Favicons are compared by source, and the screenshots are compared

visually. Each screenshot is divided in a raster of fixed-size tiles (e.g., 10x10 pixels). Each tile is compared to its counterpart in the stored snapshot data. If the tiles do not match exactly, the area covered by it is marked as changed. The rastering and comparison algorithms are implemented using the recently introduced HTML5 canvas element, which offers extensive image manipulation capabilities.

One potential disadvantage of the screenshot analysis is the difficulty to detect a small change in a page that results in a visible shifting of contents (e.g. adding one message in front of a list). Such *false positives* may be addressed by a smarter comparison algorithm, that is able to detect movements within a screenshot.

The evaluation section (Section 6.4) discusses the chosen tile size and performance of the comparison algorithm in more detail.

**Highlighting Differences.** Once the differences for a focused tab are calculated, TabShots injects an overlay into the page. This overlay is completely transparent, except for the differences, which are shown in semi-transparent red. The overlay is positioned in the top left corner and covers the entire visible part of the site. Setting the CSS directive *pointer-events: none* ensures that the overlay does not cause any unwanted interactions, and allows mouse and keyboard events to “fall through” the overlay onto the original content.

In order to detect a malicious page from actively trying to remove the overlay from the DOM, we implement a mutation event listener that is triggered when an element is removed. It then checks whether the overlay is still present and if not, immediately warns the user of this active malicious behavior.

**Security Indicator.** In addition to the overlay of the changes on the current page, TabShots also has a browser toolbar icon, indicating the current status of the site. The icon’s background color indicates how much of the site has changed, ranging from almost nothing (< 10%, shown as green), over moderate (< 40%, shown as yellow) to high (> 40%, shown as red). Clicking on the icon shows a miniature view of the current tab combined with the overlay of detected changes. Having a security indicator as part of the browser environment ensures that even if a malicious page somehow manipulates or removes the overlay, the user still has a trustworthy notification mechanism.

The current notification mechanism is quite subtle, but follows other commonly accepted and implemented notification mechanisms, such as displaying a padlock when using a secure connection. If desired, the notification mechanism can be

easily extended to something more visible, such as the warnings given in case of an invalid SSL certificate.

### 6.3.3 Alternative Design Decisions

During the design and development of TabShots, we considered different paths and options, leading to the outcome described here. For completion, we want to discuss two topics that drove the design and workings of TabShots in a bit more detail.

**JavaScript-Based Detection.** Instead of visually comparing screenshots, we might attempt to detect the malicious JavaScript code actually carrying out the tabnabbing attack. This is not a trivial task, since JavaScript's dynamic nature makes script analysis difficult. Furthermore, there are a multitude of ways of actually implementing a tabnabbing attack. The attack example discussed earlier uses the `window.onBlur` event, but a tabnabbing attack is certainly not limited to only this event. Similarly, there are numerous ways of actually changing the displayed content, ranging from the use of JavaScript to extensive use of available CSS techniques.

**Regularly Capturing Tabs.** Currently, TabShots makes a capture of a tab at regular intervals, so it can compare the capture taken when the user returns to a fairly recent capture from before. Ideally, we would make a capture when the user leaves, and a capture when the user returns. Unfortunately, Chrome does not trigger an event when a user leaves a tab, only when a user focuses a new tab. At the moment this event is received, the new tab is already displayed. To take a screenshot of the tab that was just left, TabShots has to switch it back into display, take a capture and switch back to the new tab. Unfortunately, this cannot be implemented without very briefly revealing this process visually to the user, with a degraded user experience as a consequence.

## 6.4 Evaluation

As discussed before, a tabnabbing attack takes place when a user leaves a innocuous-looking malicious tab unfocused. Tabnabbing is different from traditional phishing, since it exploits trust placed in a previously opened tab, whereas phishing simply tries to mislead the user.

Our evaluation of TabShots consists of three parts. First we discuss how TabShots effectively protects against all tabnabbing attacks. Second, we discuss the performance impact of TabShots. The third part elaborates on the setup and results of an experimental compatibility study using Alexa's 1,000 most popular sites.

### 6.4.1 Security

The security guarantees offered by TabShots follow directly from its design. We recapitulate the three most important security properties here: (i) zero false negatives, (ii) user-friendly and clear overlay and (iii) secure toolbar indicator.

TabShots cannot miss a tabnabbing attack by design, since it visually captures screenshots from a tab and compares them. In order for a tabnabbing attack to occur undetected, it has to ensure that the screenshots before and after losing focus are identical, meaning the page did not change while out of focus. This case is considered a classic phishing attack, and not a specific tabnabbing attack.

Second, TabShots injects an overlay of the focused tab, indicating which parts of the page have changed since its last focus. Using mutation events, TabShots detects if a malicious page actively tries to remove the overlay, and notifies the user with a strong security message.

Third, TabShots also adds an icon to the browser toolbar. Using a three-level color indication system, it notifies the user of how much a tab did change. The strength of this toolbar icon is that it runs in the context of the add-on, and is completely out of reach to any page-specific code. This effectively prevents any manipulation by a malicious page.

### 6.4.2 Performance

In order to prevent tabnabbing attacks, TabShots must be capable of warning the user of any changes *before* she enters any sensitive information. Furthermore, since TabShots's algorithm is executed when a user switches tabs, it is crucial that there is no noticeable performance impact. The performance measurements and analysis of the main algorithm, discussed below, show that TabShots succeeds in quickly processing the captures and warning the user of any changes that occurred.

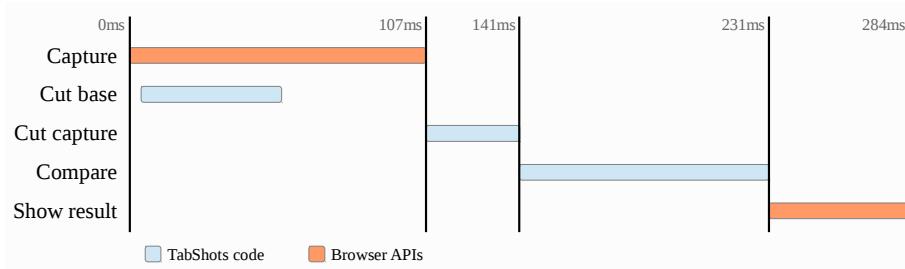


Figure 6.5: Breakdown of the average performance with a resolution of 1366x768.

One important advantage of TabShots is that it fully operates in the background, without any blocking impact on any browser action or processing. When a user switches tabs, TabShots will perform the following steps:

1. Capture a screenshot of the newly focused tab
2. Cut the previously captured image of this tab (before it lost focus) into tiles
3. Cut the newly acquired screenshot into tiles
4. Compare the tiles of both screenshots and mark the differences
5. Inject the calculated overlay into the page and update the TabShots icon

For a browsing window with a resolution of 1366x768, the most common resolution at the time of this writing [228], TabShots is capable of performing these steps within an average time of 284ms after receiving the browser event fired by switching tabs. Fig. 6.5 shows a breakdown of this time into the steps mentioned before. Note that of these 284ms, 160ms are consumed by browser APIs, which are out of our control.

Currently, a large chunk of time is consumed by the comparison algorithm, which is a pixel-by-pixel comparison of each tile. The time used by this algorithm is strongly correlated to the number of changes within a page. If a difference between tiles is detected at the first pixel, there is no need to check the remaining pixels. Consequently, if a tabnabbing attack occurs, a lot of changes will be detected and TabShots's algorithm will perform even faster. Table 6.1 presents the number of milliseconds spent on comparison on our testing pages, where we use a div to change a certain percentage of a page, clearly showing the correlation between amount of changes and required processing time.

Table 6.1: Correlation between amount of changes on a page and number of milliseconds consumed by the comparison algorithm.

% changes	ms spent on comparison
0	126
25	86
50	60
75	32
100	4

Overall, one can see that TabShots is efficient enough to prevent tabnabbing attacks, before the user discloses her credentials to the phishing page and without a negative effect on the user’s browsing experience. Moreover, if TabShots was to be implemented directly within the browser instead of through the browser’s add-on APIs, we expect that its overhead would be significantly lower.

### 6.4.3 Compatibility

Apart from the security guarantees offered by TabShots, its compatibility with existing sites is another important evaluation criterion. When using non-malicious Web applications, the number of changes detected by TabShots, i.e. false positives, should be limited, even though the user can quickly determine whether a change is legitimate or not.

To determine the compatibility with current Web applications, we ran TabShots on the top 1,000 Alexa sites. Each site was loaded in a separate tab, and captured before and after it lost focus. These two captures were compared and analyzed for the number of changed blocks. Through our preliminary experimentation with TabShots, we discovered that a 10x10 tile-size strikes the desired balance between performance and precision. Smaller tiles would incur extra overhead, since as the number of tiles increase, so do the checks between the old versions and the new ones, without a distinguishable improvement in pin-pointing the modified content.

Table 6.2 shows the results for the top 20 sites, and Figure 6.6 shows a histogram of the entire top 1,000, grouped by integer percentage values. The results show that 78% of sites fall within the safe threshold of less than 5% changed blocks, meaning there are no compatibility issues here. About 19% of sites have moderate changes, but still less than 40%. Manual verification shows that these changes are mainly caused by changing content such as image slideshows

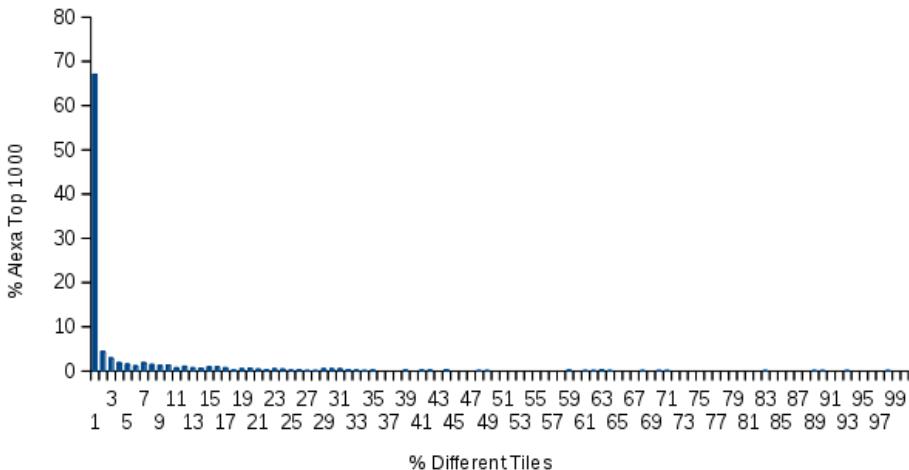


Figure 6.6: Compatibility analysis of the visual comparison algorithm with Alexa’s top 1,000 sites.



Figure 6.7: Before and after shots of [americanexpress.com](http://americanexpress.com) (#354), which has 38.93% of changed blocks, due to a background image that took longer to load.

or dynamic advertisements. A typical example of an overlay of a dynamic advertisement is shown in Figure 6.9. Finally, 3% of sites has more than 40% of changed blocks, which seem to be caused by changing background graphics. Figure 6.7 and Figure 6.8 respectively show the worst case scenario for the sites with moderate changes (less than 40%) and sites with heavy changes (more than 40%).

Note that even though certain sites have a high number of changed blocks, TabShots never interferes with a page, preventing any loss of functionality. If desired, a user can easily whitelist known trusted sites, to prevent needless overlaying of changed content. Additionally, a future extension of TabShots can incorporate a learning algorithm to identify dynamic parts of a site while the tab is in focus, which reduces the number of false positives.

Table 6.2: Compatibility analysis of the Alexa top 20 sites

Domain	% of changed tiles
facebook.com	0.38
google.com	0.00
youtube.com	4.05
yahoo.com	5.31
baidu.com	0.00
wikipedia.org	0.73
live.com	2.65
twitter.com	2.91
qq.com	6.00
amazon.com	2.57
blogspot.com	0.32
linkedin.com	0.26
taobao.com	0.49
google.co.in	0.00
yahoo.co.jp	4.13
sina.com.cn	1.24
msn.com	23.22
google.com.hk	0.00
google.de	0.00
bing.com	0.00

Figure 6.8: Before and after shots of *mlb.com* (#355), which has 97.31% of changed blocks, due to an overlay that changed the intensity of the site to present an advertisement.

The automated analysis gives a good idea of the impact on Alexa's top 1,000, but is unfortunately not able to cover the authenticated parts of the sites. Therefore, we also tested the impact of TabShots on the daily use of several highly dynamic Web applications, for example social networking applications (e.g. Facebook, Twitter) and Webmail clients (e.g. GMail, Outlook Web Access). One noticeable effect is that the addition of a single element can cause a shifting

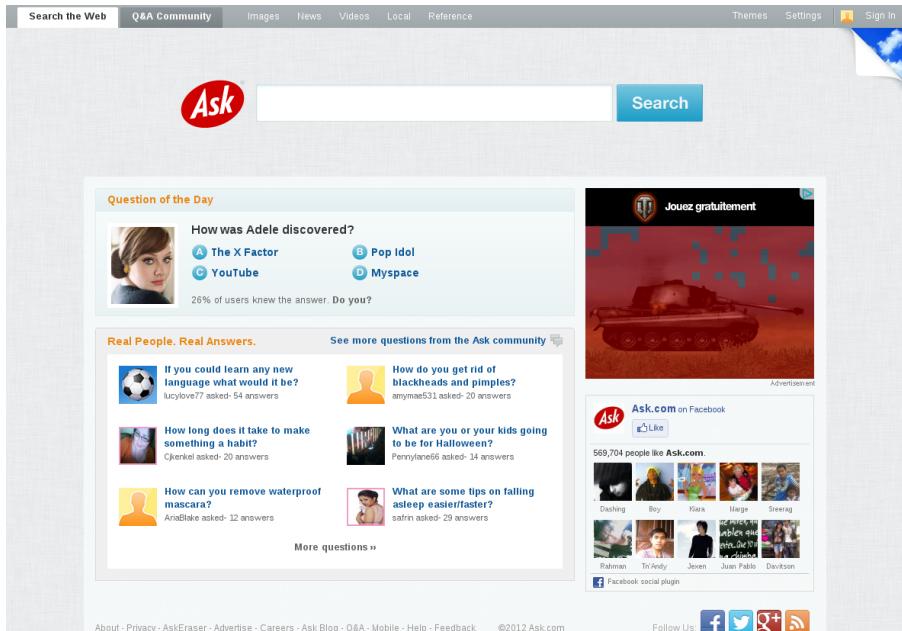


Figure 6.9: Screenshot of a typical dynamic advertisement being recognized by TabShots.

of content within a page, which is currently flagged as a major change by the comparison algorithm. In future work, we can implement a comparison algorithm that detects such shifts and only marks the newly added content as a change.

## 6.5 Blacklisting Tabnabbing

In the previous section, we described in detail the idea and implementation of TabShots. While this is sufficient for the protection of a user who has installed our browser add-on, we deem it desirable to also protect users who are using different browsers or have not installed TabShots. This can be achieved through an optional server-side component which can aggregate information sent by individual browsers and, after validation, add the reported URLs in a blacklist. This server-side component would be the logical next step, to transition from a protection of a selected number of users (those with TabShots installed) to a more global protection, similar to Google's SafeBrowsing list of malicious

sites [76] which is currently utilized by many modern browsers. In the rest of this section, we describe the possible workings of such a service.

In the stand-alone version of TabShots, once a user realizes that she is being targetted by a tabnabbing attack, she is instructed to simply navigate away from the malicious site without entering any private information. With a server-side component in place, the user can mark the current page as a “tabnabbing attack” through the UI of our add-on. Once this happens, TabShots transfers to a server-side, data aggregator the following information:

1. The URL of the current page
2. The image of the page before the user switched tabs
3. The image of the page after the user switched tabs

The server-side aggregator has the responsibility of receiving reports from multiple users, filtering-out false reports and then adding the true positives on a blacklist. Filtering is necessary to stop attackers who wish to harm the reputation of the TabShots blacklist, by submitting legitimate sites that would then be automatically blocked. Our server-side service operates as follows:

For every previously unreported URL received by a user with TabShots installed, our service spawns an instrumented browser which visits the reported URL and captures a screenshot. Assuming that the current page is indeed performing tabnabbing, the malicious scripts will try to get information about their “visibility” through the `window.onBlur` event. Since our browsers are instrumented, we can trigger a `window.onBlur` event without requiring the actual presence of extra tabs. In the same way, any callbacks that the script registers using the `setTimeout` method, are immediately triggered, i.e., the malicious code is tricked into performing the tabnabbing attack, without the need of waiting. Once the callbacks are executed, our system takes another snapshot of the resulting page. The set of screenshots captured by the user is then compared with the set that was captured by our system. To account for changes in the pages due to advertisements and other legitimately-dynamic content, the screenshots are accepted if either the server-generated set is an identical copy of the user-generated one, or if they match over a certain configurable threshold (i.e. everything matches except certain dynamic areas).

Once the above process is complete, the URLs recognized as true positives are then sent to a human analyst who will verify that the resulting page is indeed a phishing page. A human analyst is necessary since our system cannot reason towards the maliciousness or legitimacy of the final changed page. Note, that human-assisted phishing verification is currently one of the most successful

approaches, e.g., PhishTank [190], and its results are more trustworthy than any automated phishing-detection solution. The URLs that are marked as “tabnabbing”, can then be added to a blacklist that browsers can subscribe to.

The users who report URLs that either never reach a human analyst (because the server-side screenshots did not match the user-provided ones), or reached a human analyst and were classified as “non-phishing” are logged, so that if they are found to consistently submit false positives, our system may adapt to ignore their future submissions.

Submitting screenshots to a third party service might be considered privacy-sensitive, so we take care to address these issues accordingly. Therefore, TabShots only submits a screenshot after explicit user approval. Additionally, the screenshot submission is only triggered after the user flagged a tabnabbing attack, so it is very likely that the captured site is malicious of nature, and does not contain any sensitive information.

## 6.6 Related work

Raskin was the first to present the tabnabbing attack in 2010 [201]. Others, presented variations of the attack, for instance redirecting the user through a meta-refresh tag to a new page, instead of changing the existing page with JavaScript [6], which would circumvent protections such as NoScript [167]. This attack however does not depend on user activity, but rather on a predefined timeout, by which the attacker hopes that the victim will have changed tabs and thus will not notice the changing Web-site.

Unlu and Bicakci [244] proposed NoTabNab, a browser add-on that monitors tabs in search for tabnabbing attacks. When a page loads, the add-on records the title of the page, the URL, the favicon and several attributes of the topmost elements,<sup>4</sup> as determined by the browser API call `document.elementFromPoint(x,y)`. While this approach is conceptually similar to ours, NoTabNab suffers from several issues that render it ineffective.

A first issue is that by capturing a tab when it is loaded, the add-on will miss all content that is added dynamically (e.g. using AJAX) between loading a tab and actually switching to another tab. Second, the design of the detection mechanism offers an attacker several ways of evading it. For instance, an attacker can place all of the page’s content in an iframe that spans the entire visible window. The `document.elementFromPoint` cannot “pierce” through

---

<sup>4</sup>When seeing a page as a stack of elements, the topmost elements potentially overlap other elements

the iframe, and will always return the iframe element, regardless of any changes that may happen inside the iframe. Another possible bypass, is through the overlay of a transparent element, that again stretches the entire page’s content and allows clicks and interactions to “fall-through” to the actual phishing form under it. On the contrary, TabShots uses the screen-capturing API of the browser and is essentially using the same data as actually seen by the user. This design decision makes TabShots invulnerable to the aforementioned bypasses.

Suri et al. [232] propose the detection of tabnabbing through the use of tabnabbing “signatures”. The authors claim that the combination of certain JavaScript APIs with HTML elements are tell-tale signs of a tabnabbing attack, and present two signatures, based on the presence of `onBlur`, `onFocus`, and other events within an iframe. Unfortunately, the authors make no attempt to characterize the false positives that their system would incur. Additionally, the presence of an iframe is by no means necessary for a tabnabbing attack. JavaScript code is capable to drastically change the appearance of a page through the addition and removal of styled HTML elements, thus allowing an attacker to bypass the authors’ monitor. TabShots on the other hand, does not depend on anything other than the visual differences between the old and the new version of the tab, and thus will detect all visible changes, regardless of the technical means through which they are achieved.

While tabnabbing is a relatively new phishing technique, attackers have been trying to convince users to voluntarily give up their credentials for at least the last 17 years [152]. Several studies have been conducted, trying to identify why users fall victim to phishing attacks [78, 88] and various solutions have been suggested, such as the use of per-site “page-skimming” [77], security toolbars [259], images [7, 21], trusted password windows [214], use of past-activity knowledge [184] and automatic analysis of the content within a page [265]. Unfortunately, the problem is hard to address in a completely automatic way, and thus, the current deployed anti-phishing mechanisms in popular browsers are all black-list based [76]. The blacklists themselves are either generated automatically by automated crawlers, searching for phishing pages on the Web [255] or are crowdsourced [190].

## 6.7 Conclusion

Tabnabbing attacks are a type of phishing attacks where the attacker exploits the trust a user places in previously opened browser tabs, by making the malicious tab look like a legitimate login form of a known Web application. This happens

when the user is looking at another tab in the browser, making it very hard to detect and very easy to fall victim to.

Currently available countermeasures typically depend on several specific characteristics of a tabnabbing attack, and are easily bypassed or circumvented. Our countermeasure, TabShots, is the first to do a fully visual comparison, detecting any changes in an out-of-focus page and highlighting them, aiding the user in the decision whether to trust this page or not.

Our evaluation shows that TabShots protects users against potential tabnabbing attacks, with a minimal performance impact. Furthermore, an experimental evaluation using Alexa's top 1,000 sites shows that 78% of these sites fall within the safe threshold of less than 5% changes in subsequent snapshots. This means that TabShots is fully compatible with these sites, and has very little impact on another 19%.



# Chapter 7

## Reflections on Client-side Web Security

The previous chapters have shown that even when traditional server-side mitigation techniques are available, application developers are often unaware or uninterested in implementing them, leaving the users to be vulnerable in the end. For example, every modern home contains a network router, notoriously vulnerable to traditional Web attacks [210], every smartphone is Web-enabled, often running vulnerable Web applications [162], and some devices even only run a browser, putting all eggs in the basket of Web security.

In order to allow users to defend themselves against these vulnerabilities, autonomous client-side countermeasures have been proposed against a long list of attacks, of which several were discussed in previous chapters. Due to the evolution of the browser into a full-fledged client-side platform, the traditional server-driven policies have also been evolving into server-driven, client-enforced policies, making the browser a center point of security in the modern Web.

In this chapter, we reflect on this field of client-side Web security, based on our experience gained during this dissertation. We first explore the different implementation options of client-side security technologies, each with their own advantages and disadvantages, and subsequently focus on the use of browser add-ons to implement research prototypes. We subsequently look into research challenges on the road ahead, and observe some trends in the current state-of-practice.

## 7.1 Experience Report on Client-Side Mitigations

The previous four chapters all introduced a client-side mitigation technique against attacks on the security of Web sessions. Because the previous chapters mainly focused on the attacks and proposed mitigation techniques, we use this section to elaborate on the implementations behind the prototypes. We first provide an overview of different implementation strategies, ranging from an OS-level application to in-browser protection by instrumenting the Web application. After this overview, we continue with a discussion of the specifics of browser add-ons, the recurring implementation strategy in the previous chapters, and expertise of this dissertation [70]. Finally, we discuss the evaluation challenges raised by client-side mitigation techniques.

### 7.1.1 Positioning a Client-Side Mitigation Technique

Client-side mitigation techniques can be implemented at various levels. Since the previous chapters have only discussed the use of browser add-ons, we first take a step back and present a broad overview of the different strategies for implementing a client-side mitigation technique.

#### OS-level Application

A first approach for implementing a client-side mitigation technique is to build a traditional OS-level application, which listens for interesting events, such as network traffic going out. The operating system offers hooks for a wide variety of system operations, such as network activity or file system access, allowing an application to run its code when such an event occurs. This approach is commonly used by antivirus products to detect a virus the moment a file is accessed.

One of the main advantages of using OS-level hooks is the independence of the countermeasure from the client application, which can be a browser, a game, or anything else running on top of the OS. An additional advantage of these OS-level hooks are the capability to operate on multiple applications simultaneously, enabling the enforcement of a cross-application security policy. The price of this flexibility is paid when a hook is triggered, since the invoked code needs to process low-level data, such as network packets, which requires a significant amount of effort. Additionally, at the level of the OS, all context information from the client application that generated the data is lost, which may be a limiting factor to implement the security policy at hand.

## Client-Side Proxy

Using a client-side proxy to implement a network-based client-side mitigation technique is a commonly used approach. The client applications send their network traffic through a proxy, a feature that is supported by most applications nowadays. The proxy enforces its security policy by inspecting and modifying the traffic that passes through. Proxies can be developed as client-side applications running on the same machine as the client application, or can be deployed on a separate device in the local network. Several research proposals successfully use the proxy-based approach to protect the client application against cross-site request forgery [143], session hijacking [185] and SSL stripping [187].

The proxy-based implementations still have the advantage of being independent from the client application, allowing a free choice of browser. Additionally, they can intercept the traffic from multiple applications, allowing the cross-application enforcement of a security policy. Compared to an application using OS-level hooks, proxies can also operate on a specific protocol, requiring only limited effort to process the network traffic, for example into HTTP requests. One important disadvantage remains the lack of context information, which may complicate the enforcement of a security policy, for example when the user is using two separate browsing sessions. Additionally, proxy-based approaches are restricted to network-based security mechanisms, which is, however, a less significant restriction in the context of the Web.

## Customized Client Application

A third approach directly integrates the mitigation technique in the client application, typically a browser. The client application's code is modified to integrate the security mechanism, enforcing the security policy from within the application. This approach is often used to implement complex security mechanisms, such as an architecture for least-privilege isolation of components in a Web page [246], the enforcement of information flow control for JavaScript [62], or the binding of cookies to TLS channels [81].

The most significant advantage of directly implementing the security mechanism in the target application is the high degree of freedom. If necessary, every aspect of the application can be modified to ensure complete mediation when enforcing the security policy. Additionally, since the mitigation technique is implemented within the client application, it can access all the required context information, such as the Web page and its window, which supports the enforcement of fine-grained security policies. One disadvantage of customizing an application is the high degree of complexity of modifying a browser's codebase.

Additionally, unless the mitigation technique will be adopted by browser vendors and integrated in mainstream browsers, wide-scale deployment across the Web is practically impossible.

### **Browser Add-on**

Users can install browser add-ons to modify and extend the default browser behavior. In this scenario, browsers offer hooks to internal browser events, to which an add-on can register custom handlers. For example, exposed hooks allow the interception of network requests and responses, the inspection of the DOM after loading, etc. Additionally, browsers expose a set of features through APIs, allowing add-ons to read and write files, send custom network requests, access the page's contents, etc. Browser add-ons are a quick and easy way to implement a research prototype, not only used by the work presented in the previous chapters, but also by a proposal to force HTTPS connections [137], by an approach to offer Web application integrity [113], and by an alternative approach to prevent CSRF attacks [212].

An important advantage of a browser add-on is the lightweight development process, enabling the quick validation of a research idea. Following up on good research ideas, browser add-ons can be used to build prototypes, as well as marketable products, as illustrated by CsFire. Since add-ons are essentially part of the browser, they can use the available context information to enforce a fine-grained security policy. This brings us to one of the disadvantages of browser add-ons: they are limited to the APIs offered by the browser. As we will discuss further in this section, Firefox offers a wide variety of APIs, even allowing to run external programs, while Chrome keeps the APIs very tight, preventing an add-on from breaking out of the browser process' sandbox, resulting in a more secure system.

### **Instrumented Web Application**

A final approach to implement a client-side security mechanism is to instrument the Web application with a security mechanism, that will be executed by the browser. In this scenario, the code of the security mechanism is embedded in the application's pages, script files and other Web content that will be distributed to the client. This approach is often used to implement technologies that isolate certain components from the application [194, 9, 237].

The main advantage of this approach is the lack of any installation procedure at the client-side, allowing the security mechanism to be distributed to all

clients at once. As this technique is an instantiation of an inline reference monitor [91], it also shares the challenges of inline reference monitors to achieve complete mediation, and avoid tampering with the code of the security mechanism. Additionally, the code of the security mechanism is restricted to the same privileges as the application code, which imposes limits on cross-origin interactions.

### 7.1.2 The Capabilities of Browser Add-ons

Since the focus of this dissertation lies on add-on based implementations, we elaborate on the capabilities that are available to add-ons, and how these differ between the two most used platforms, which are Mozilla Firefox and its mobile counterpart, Fennec, and Google Chrome's desktop version of the browser.

Browsers expose numerous feature-rich APIs towards browser add-ons, granting these add-ons rich mediation power, deep-reaching access to the loaded pages, and the capability to extend the browser's UI. In summary, these three capabilities are necessary to build a client-side mitigation technique, for the following reasons:

- **Mediation Power:** Add-ons possess rich mediation power through the browser-provided hooks. A commonly used example of these hooks are the network events, where HTTP requests and responses are intercepted, allowing an add-on to inspect and modify the traffic, or to block a request from leaving the browser.
- **Page Access:** Add-ons can access the contents of a loaded Web page, allowing them to not only inspect the DOM tree of the document, but also to modify the tree by injecting and deleting nodes. As an extreme example, the Greasemonkey add-on even allows users to define their own scripts to manipulate the DOM of certain sites, thereby introducing new security vulnerabilities [247].
- **UI Modifications:** Add-ons have the possibility to modify and extend the browser's UI, enabling them to interact with the user using additional toolbars, buttons, menu items, etc.

How these capabilities are exposed depends on the browser's underlying architecture, which vastly differs between browsers. Firefox supports privileged add-ons, which not only have access to a large set of browser-provided APIs, but also to the browser's internals and operating system resources, enabling capabilities such as reading/writing files, spawning new processes, etc. In Firefox,

add-ons all live within the same namespace, and can define core components with an API to be exposed to other add-on, as well as to scripts that interact directly with Web content. In the recently introduced JetPack model [180], add-on can also choose to adopt a more modular model, offering some isolation and restrictions, even though the rich APIs remain accessible if desired. Concretely, add-ons in Firefox are subject to very few limitations, and can easily share their functionality among core components and scripts interacting with Web content.

Chrome is based on the principles of least privilege, isolated worlds and permissions [25, 50]. Add-ons have a core component, which runs separately from content scripts, which interact with actual Web content. Communication between both contexts is only available through the Web Messaging API [115]. Additionally, add-ons all have a distinct namespace, and are isolated from each other. The browser APIs offered by Chrome are more limited than the Firefox APIs, and do not support reaching out of the browser sandbox, into the OS. Furthermore, Chrome add-ons have to explicitly request a set of permissions for a determined set of Web sites (wildcards are allowed) upon installation. Without the necessary permissions, several APIs become inaccessible, preventing an add-on from escalating its power within the browser.

We initially implemented CsFire on the Firefox platform, and later expanded support towards Google Chrome as well. Both platforms offer several hooks to inspect and manipulate network requests as they leave the browser, allowing the enforcement of CsFire’s security policy. To optimize the use of shared code between both platforms, CsFire’s core functionality is implemented in a platform-independent manner, and uses platform-specific adapters towards the browser-provided APIs. These adapters invoke the correct APIs, register events to the correct hooks, and translate between generic objects and browser-specific objects, such as the internal representation of a URI.

Clearly, the features offered by Firefox open a wide range of possibilities for browser add-ons, but these possibilities come at the price of security guarantees, which are virtually non-existent in the Firefox model [25]. Chrome, on the other hand, confines the add-ons within the browser, preventing them from compromising the client device running the browser.

**Conclusion.** In our experience developing numerous browser add-ons for both Firefox and Chrome, we can conclude that the Chrome platform is the easiest to get started with. Chrome offers clear APIs and a straightforward system to register to internal browser events, as well as a developer console for inspecting and debugging the add-on code in real time. Firefox, on the other hand, offers a more complex system to register to browser events, and the internal browser APIs are much more complicated. On the plus side, Firefox offers significantly

more freedom, allowing an add-on to easily call native libraries or even execute arbitrary programs on the operating system. Unfortunately, Firefox does not offer easy-to-use debugging tools, making the development process more difficult.

### 7.1.3 Evaluating Browser Add-on Prototypes

Building research prototypes also involves an extensive evaluation, to show the benefits of the proposed mitigation technique, as well as the potential impact. We report on our experiences with both large-scale evaluations on the Alexa top 1,000,000, and smaller-scale evaluations on the Alexa top 1,000.

One aspect of evaluating a client-side mitigation technique is determining its effectiveness at mitigating the targeted attacks. For our prototypes, we have built test cases, which effectively attempt to carry out the attack in a lab setup, allowing us to verify whether the mitigation technique actually works or not. For *CsFire*, we even built a full security benchmark, listing all CSRF attack vectors in the HTML and CSS specifications. Such a benchmark is beneficial for the security evaluation of a prototype, allowed us to perform rigorous testing before releasing a new version, and has been subsequently used in a master's thesis on building an automated platform for testing the effectiveness of client-side countermeasures.

A more challenging aspect of evaluating autonomous client-side mitigation techniques is determining the compatibility with current sites. Since these countermeasures operate independently from the Web applications loaded in the browser, they must aim for complete compatibility. While this goal is practically never fully achieved, all evaluations do focus on the compatibility aspect, carefully discussing the false positives that occur. Gathering data, both about the presence of vulnerabilities as the effectiveness of the countermeasure at hand, is extremely important, but also very challenging. For example, the experimental evaluation of *CsFire* is only possible because of the cooperation of numerous subjects who willingly recorded information about their browsing traffic. For *Serene*, the use of heuristics to identify session identifiers was necessary, which has inescapably led to false positives. Additionally, detecting the presence of vulnerabilities like CSRF requires interaction with the Web application at the server side, not something to be run in an automated fashion on a large scale, as actions may have unexpected consequences. Similarly, wide-scale detection of session fixation and session hijacking vulnerabilities requires automation of Web application's authentication processes, not a trivial task. Being able to explore the authenticated parts of Web applications would be a breakthrough for the field of Web security, both for exploratory research efforts as for the evaluation of newly proposed countermeasures. Automating

arbitrary authentication procedures, requiring dedicated, application-specific accounts, is a strong challenge. A slightly weaker, but almost equally useful challenge can take advantage of the prevalence of social networks, as numerous applications nowadays offer social login systems, where a single social networking account grants access to a wide variety of applications. An automated discovery process of applications that support social login, should enable the automated, authenticated crawling of such applications.

Since evaluating a client-side countermeasure generally involves visiting a large amount of pages, automating the procedure is recommended. Specifically for a prototype built as a browser add-on, the evaluation setup requires control over an actual browser to visit the pages. Depending of the number of pages that need to be visited, the evaluation setup can run on a single machine in a matter of days, or may require a more elaborate, distributed setup.<sup>1</sup> We have used both, a single-machine setup for the evaluation of *TabShots* on the Alexa top 1,000, and a distributed setup for *Serene*, where we visited the Alexa top 1,000,000. Fortunately, ample tools are available to control the browser remotely, passing it URLs of pages to visit. One approach is to directly invoke the browser executable, and pass it a URL, as we did with *Serene*. A second approach is to use an add-on, such as ChickenFoot [40], to instruct the browser to visit a list of URLs, an approach we followed with *TabShots*. A third approach is to use a browser automation framework, such as Selenium, to actually control the browser and inspect the resulting page. We used the latter approach to implement a set of unit tests for *CsFire*, both to evaluate its effectiveness against a large amount of CSRF attack vectors, as to perform pre-release unit and regression tests on every subsequent release, where additional features were added.

Finally, from our experience in demonstrating and describing the use of add-ons for building prototypes, we have learned that the nuances of add-ons, and the differences between an add-on and browser customizations are not always fully understood. Therefore, the use of add-ons to build research or demonstration prototypes warrants the careful explanation of the drawbacks and benefits.

## 7.2 Research Challenges and Trends

The field of Web security is in constant evolution, both in research as in practical deployment scenarios. At the end of this dissertation, we look into

---

<sup>1</sup>For more information about distributed setups, we refer you to the PhD of Steven Van Acker [245], who is the driving factor behind these setups, and was always willing to assist.

future research challenges within this field, as well as into currently observed trends in the state-of-practice.

### 7.2.1 Theoretical Underpinnings of Web Security

Due to the complex nature and enormous size of the constantly evolving Web, security research is often of a very practical nature, for example to understand the different ecosystems within the Web, or to develop and evaluate new mitigation techniques, as illustrated by the previous chapters of this dissertation. However, the prevalence of this practical line of research does not preempt the importance of more theoretical approaches to Web security, which employ formal modeling techniques or rigorous analysis methodologies to propose new, secure-by-design technologies or evaluate the security of currently proposed or deployed systems.

Within this doctorate, we have gained experience with both approaches. In Chapter 3, we have already shown how a model checker can be used to evaluate the effectiveness of CsFire’s security policy. Additionally, we have performed a security analysis of 13 mature W3C specifications, including HTML 5, as requested by the European Network and Information Security Agency (ENISA) [66, 69]. From the vast amount of specification text (more than a 1,000 A4 pages), we distilled the functional capabilities offered by the specification, how the user is involved in the security model, and what implicit or explicit security and privacy assumptions are made by the specification. Based on this information, we have performed a rigorous threat assessment, focused around the threat models of a *Web attacker* and a *gadget attacker*. We have uncovered 51 security issues in total, of which we categorized 6 as severe, which included for example the possibility to tamper with a form using an HTML injection attack, or a violation of the semantic model of the Web by the CORS specification [248]. The others could mainly be attributed to inconsistencies between the specifications, as well as to lack of precision when specifying security requirements or features. Additionally, the study showed an increasing reliance on origin-based permission models, which not only puts the security responsibility on the user, but also clashes with the lack of isolation between integrated scripts inside a page.

Based on our experience with both tracks within the theoretical approaches towards Web security, we can make two important observations. In a first observation, we acknowledge the importance of theoretical approaches for offering strong security guarantees, especially the formal verification of desired security properties and proposed security policies. However, the main challenge in successfully applying these approaches lies in overcoming the discrepancies between the formal model and the actual, practical implementations. Building a

formal representation of certain aspects of the Web requires making the necessary abstractions, in order to make the modeling effort both feasible and scalable. Unfortunately, when going from theory to practice, the implementation will need to take care of the details that have been abstracted away, which often results in insecure or incomplete implementations. A common illustration of these difficulties can be found in the formal approaches towards JavaScript security, where some of the less-elegant language features are abstracted away by means of a well-defined subset. While making these abstractions is extremely useful in supporting the theoretical work on JavaScript security, it also represents the difficulty of applying such approaches in the real world.

In a second observation, we would like to stress the research value in performing a security analysis on existing systems. The results of such a security analysis can be used to improve the security of these systems, or to guide the development of alternative systems. One example is our security analysis of next-generation Web specifications, where amendments to the specifications have been made, based on the threats and issues that have been uncovered. Another example is Google Chrome's add-on system, which was developed based on the results from an analysis of Mozilla Firefox's add-on system [25]. A second security analysis of the first add-on architecture of Google Chrome has led to additional security improvements, such as the mandatory enforcement of Content Security Policies on new add-ons [229].

### 7.2.2 Upcoming State-of-Practice Security Policies

An important trend in client-side Web security is the rise of *server-driven client-side security policies*. These policies have been developed alongside the line of research on autonomous client-side mitigation techniques, both in research, as illustrated by the WebSand project,<sup>2</sup> as in practice, as illustrated by examples such as Content Security Policy [229]. These server-driven security policies are delivered by the server, where the desired security properties and underlying application structure is known, and are enforced by the browser, the most optimal location for enforcing client-side security policies.

Server-driven client-side security policies have emerged from a constantly evolving security process, which started with generic browser security policies, such as the Same-Origin Policy, which were applied to all applications, without exceptions or customizations. As the server-side processing component of Web applications became more advanced, Web security mechanisms and countermeasures against concrete attacks were to be implemented at the server-side. Well-known examples are token-based approaches against CSRF attacks,

---

<sup>2</sup>More information can be found on <https://www.Websand.eu>

or input and output sanitization against injection attacks. While these security mechanisms initially required a significant amount of effort from the developer, the developer community quickly caught up, offering easy-to-use libraries and out-of-the-box support in popular development frameworks. Unfortunately, as we witness up until today, merely offering support for the appropriate security technology is not sufficient to secure the Web. Many developers are not aware of all the security technology they need to incorporate, and legacy applications are often not updated to include the optimal security solutions, due to various reasons, such as technical difficulties or a lack of interest.

In response to these applications remaining vulnerable, while adequate mitigation techniques being available, a movement of autonomous client-side countermeasures started to offer users the protection they desired. A popular example is the NoScript browser add-on [167], whose main feature is preventing JavaScript from executing, and is installed by more than two million Firefox users. Autonomous client-side countermeasures operate without support of the applications they protect, and need to be able to support all applications their typical user visits, which is an unbounded subset of the entire Web. In order to achieve maximum compatibility, these countermeasures push the limit of what is possible, while maintaining the delicate balance between usability and security. This presents a difficult challenge, as it is not always possible to make a correct security decision without additional server-side information. For example, from *CsFire*'s point of view, a legitimate cross-origin request and a carefully crafted CSRF attack technically look exactly the same, with the only difference between them that the former is *intended* by the application, and the latter is not. Another example are the heuristic algorithms to determine whether a specific cookie value is a session identifier or not [185, 74], which are not perfect, but require manual intervention to improve [49].

As the importance of the client platform increased, new client-side countermeasures were introduced to enforce security policies provided by the server-side application, which is best positioned to provide information about the intended behavior, the important tokens and the required resources. Essentially, autonomous client-side countermeasures already supported this practice in the form of centrally administered policy databases, to minimize the impact of potential false positives. A first example of an application-driven security policy are the *HttpOnly* and *Secure* cookie flags, telling the browser which cookies can be made accessible from JavaScript, and which cookies can be sent over a non-secure channel. Another, more complex example is Content Security Policy [229], a policy initially intended to combat script injection attacks, which is now rapidly evolving into the all-inclusive client-side security policy [28], with future features such as UI security directives [169] and Entry Points [206].

An often heard critique on server-driven client-side security mechanisms, such

as the *sandbox* attribute [36] or Content Security Policy [229], is that they only mitigate part of the attacks, lack a certain level of fine-grainedness, and do not provide an airtight solution against a specific attack. Taking into account that most attacks do in fact originate from server-side vulnerabilities, such as a failure to properly sanitize input and output, this seems an unfair evaluation. These countermeasures are intended to give a security-conscious Web developer an additional tool to squash attackers that manage to evade the server-side defenses, as well as to add a layer of protection to a legacy application, which can not easily be modified to incorporate the appropriate defenses.

The seemingly secondary role of server-driven client-side mitigation techniques should not be mistaken for a sign of inutility, as they are essential for deploying an in-depth defense strategy. Applying multi-layered defenses attributes to well-established security principles, and effectively helps stopping an attacker who has managed to circumvent alternative protection mechanisms.

### 7.2.3 The Rise of the Mobile Web

The mobile Web has seen a big leap forward, driven by the rising popularity of smartphones and tablets, and the increasing speed of mobile networking technology. Contemporary Web applications offer alternative versions of their content, tailored specifically towards mobile devices. Popular features are aimed at mobile users, with location-based services as a prime example. Under the hood, numerous new technologies have been proposed and implemented to accommodate mobile applications. Examples are the Mobile CSS specification [42], the Geolocation API [196], support for accessing native device features [120] and enabling the offline use of Web applications [35, 118, 176]. However, the mobile Web is more than merely this version of the Web tailored towards users of mobile devices. The use of mobile apps, prepackaged applications that enhance the functionality of a mobile device, has pushed the use of the mobile Web even further. In the beginning of 2014, U.S. users spent more time accessing the Web using mobile apps than traditional PCs [191]. A breakdown of these results shows that 47% of Internet usage came from mobile apps, 8% from mobile browsers, and 45% from PCs.

Mobile apps are particularly interesting from a Web security point of view, since many of these applications heavily depend on Web technology, such as HTML5, CSS and JavaScript. There are two development and deployment approaches where this dependency on Web technology manifests itself. In a first approach, the entire app is built using Web technologies, and is launched within a native component capable of rendering Web pages. The major advantage of this approach is the possibility to keep the app's codebase platform independent.

Table 7.1: The list of OWASP’s Top Ten Mobile Risks [110]. Gray colored rows show similarities with OWASP’s Top Ten Project of traditional Web application security flaws [257]

1	Weak Server Side Controls
2	Insecure Data Storage
3	Insufficient Transport Layer Protection
4	Unintended Data Leakage
5	Poor Authentication and Authorization
6	Broken Cryptography
7	Client Side Injection
8	Security Decisions Via Untrusted Inputs
9	Improper Session Handling
10	Lack of Binary Protections

Development frameworks are available to package the app’s codebase into a platform-specific app, which can be installed by users. In the second approach, developers build a native app for the mobile platform, but have the capability to incorporate a component capable of rendering Web content. Commonly encountered use cases are the integration of advertisements in a mobile app, or the rendering of user documentation. The advantage of this approach lies in the combination between native code and Web content, where the former is typically faster [57], and the latter platform-independent and widely available.

The security model of mobile apps differs slightly between the different mobile operating systems that are available, but generally speaking, mobile apps are fully isolated from one another. A restricted communication mechanism is available to enable inter-app communication, such as the inter-process communication mechanism on Android [16]. Apps are confined within their runtime environment, but can access APIs and system features by requesting explicit permissions from the user. While this security model offers certain basic guarantees, it does not prevent traditional Web security problems, which have been covered extensively in the previous chapters, from propagating to mobile apps. This phenomenon is aptly illustrated by the OWASP Top Ten Mobile Risks [110], shown in Table 7.1, where numerous similarities with the Top Ten for Web applications (Table 1.1 on page 6) can be observed. Even worse, since mobile apps tend to have access to numerous device-specific features, such as sending text messages and making phone calls, and a lot of user-related data, such as contact information, the consequences of an attack can stretch far beyond the boundaries of the app under attack.

Web technologies are a fundamental building block of mobile apps, and are likely to remain so for the foreseeable future. Therefore, it is crucial to invest

in the security of these platforms, both in research and in the state-of-practice. For example, a recent study by Luo et al. [162] has uncovered security flaws in the WebView technology, a platform-specific component used to display Web pages on Android and iOS. One class of attacks allows a malicious app to attack a Web page loaded in a WebView container, for example a malicious app targeting the Facebook Web application. The second class of attacks consists of a non-malicious application being attacked by a malicious page loaded in a WebView container, for example an application showing a malicious advertisement. Other noteworthy research results include a detection tool for these WebView vulnerabilities [54], a study on unauthorized origin crossing on mobile platforms [252], and an analysis of unsafe and malicious dynamic code loading practices in Android apps [195].

# Chapter 8

## Conclusion

Over the past decade, the momentum of the Web has risen to unpredicted levels, driven by technology, connectivity, mobility and an unsatisfiable hunger for information and sharing. The maturing of the Web and its underlying technologies have enriched the platform towards a distributed application environment, where it provides an attractive alternative to stand-alone applications. Nowadays, a vast amount of features and functionalities are available at the client-side, which in turn enables a new approach to security, where security policies and enforcement mechanisms are shifted towards the client.

In this dissertation, we have focused on the importance of client-side Web security, which aims at preventing attackers from exploiting security vulnerabilities from the client-side, often causing harm to the end users. Such attacks are commonly launched from within the user's browser or any of the loaded contexts, networks close to the user, or even the user's machine. In Chapter 2, we have introduced the relevant client-side concepts, and defined the different threat models, each with their own capabilities. We have provided an outline of common client-side attacks and their mitigation techniques, with a specific focus on the security of Web sessions and session management mechanisms. We refer to our primer on client-side Web security [67], created within the context of the STREWS project,<sup>1</sup> for a more complete coverage. We have concluded Chapter 2 by focusing the scope of this dissertation, which lies with session management problems in modern Web applications. Concretely, we have defined the following threats to Web sessions:

---

<sup>1</sup>More information can be found on <https://www.strews.eu>

- **Violating session integrity**, where an attacker tricks the application to perform actions in the user's name, without taking full control of the session. Executing this threat means tricking the user or the user's browser into carrying out unintended actions within the target application.
- **Unauthorized transfer of a session**, where an attacker succeeds into transferring an established session from or to the user's browser. This essentially gives the attacker full control over the user's authenticated session, granting the attacker the same privileges as the user has.
- **Impersonating the user by establishing a session**, where an attacker gets hold of the user's authentication credentials, allowing him to establish a new authenticated session. This not only grants the attacker full control over the user's session, but also enables him to bypass any re-authentication checks that are built-in as a security measure against unauthorized session transfers.

The four technical contributions of this dissertation (chapters 3 to 6) actively counter these threats, either as an autonomous, client-side countermeasure, or by fundamentally upgrading the session management mechanism of Web applications. Each of these solutions is thoroughly evaluated, both on effectiveness against mitigating the relevant threat, as on the compatibility with current Web applications. To summarize, we have made the following four technical contributions:

- **CsFire** protects against cross-site request forgery (CSRF), the main attack vector for *violating session integrity*. CsFire applies an advanced request filtering algorithm on cross-origin requests from within the browser. CsFire is our most mature prototype, has been publicly released as an add-on for the Firefox and Chrome browser, and is used daily by thousands of users.
- **Serene** prevents session fixation attacks, a common attack vector for *transferring a session without authorization*. Serene's core security mechanism essentially prevents a fixated session identifier from being sent by the browser, and is supported by a heuristics algorithm to identify the session identifier in a set of cookies or parameters. This design effectively allows the finetuning of the heuristics algorithm, which improves the preciseness of the countermeasure. We have created a prototype implemented for Serene as a browser add-on for Firefox.
- **SecSess** proposes a new session management mechanism, essentially a fundamental approach at preventing the *unauthorized transfer of a session*. SecSess is designed to be compatible with the request flow used by current

cookie-based session management mechanisms, allowing it to be adopted as a drop-in replacement. SecSess also supports a gradual adoption path, with a fallback to currently used cookie-based session management mechanisms. We have implemented SecSess as a research prototype, with the client-side component as a Firefox add-on, and the server-side as a middleware for the Express framework on NodeJS.

- **TabShots** is capable of detecting tabnabbing attacks, a variation of phishing attacks, which are used to trick the user into disclosing his credentials. Such an attack allows the attacker to *impersonate the user by establish a new session*, a very dangerous and highly lucrative attack. Phishing attacks cannot be detected from the server side, and are designed to visually confuse the user. Therefore, we have designed TabShots to take a visual approach, based on the page as seen by the user. TabShots is implemented as a prototype add-on for Chrome.

Chapter 7 reflected on the field of client-side Web security. We shared our experiences with designing, implementing and evaluating client-side countermeasures. We gave an overview of the available implementation strategies, and zoomed in on the use of add-ons, their advantages and their limitations, followed by a description of our experiences with the evaluation of client-side countermeasures, employing security benchmarks, and small-scale and large-scale empirical evaluation techniques. We also identified several important trends and research challenges in the field of client-side Web security. These include the importance of theoretical approaches towards Web security, the relevance of upcoming state-of-practice security policies as a second line of defense, and the rise of the mobile Web, with Web technologies as a crucial building block for omnipresent mobile apps.

We can conclude that the contributions presented in this dissertation have clear practical applicability, while pushing autonomous client-side countermeasures to their limits, which requires a careful balancing of usability and security. Our contributions have sparked further research, where the security policies have been expanded towards a wider coverage, or further refined to improve precision.



# Bibliography

- [1] Privoxy. *Online at <http://www.privoxy.org>* (2013).
- [2] RefControl. *Online at <https://addons.mozilla.org/en-us/firefox/addon/refcontrol/>* (2013).
- [3] ABOBA, B., SIMON, D., AND ERONEN, P. Extensible authentication protocol (EAP) key management framework. *RFC Proposed Standard (RFC 5247)* (2008).
- [4] ACAR, G., JUAREZ, M., NIKIFORAKIS, N., DIAZ, C., GÜRSSES, S., PIJSESENS, F., AND PRENEEL, B. Fpdetective: Dusting the web for fingerprinters. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)* (2013), pp. 1129–1140.
- [5] ADIDA, B. Sessionlock: securing web sessions against eavesdropping. In *Proceedings of the 17th international conference on World Wide Web (WWW)* (2008), pp. 517–524.
- [6] ADLER, E. Tabnabbing without JavaScript . *Online at <http://blog.eitanadler.com/2010/05/tabnabbing-without-javascript.html>* (2010).
- [7] AGARWAL, N., RENFRO, S., AND BEJAR, A. Yahoo!'s sign-in seal and current anti-phishing solutions. In *Web 2.0 Security and Privacy (W2SP)* (2007).
- [8] AGGARWAL, G., BURSZTEIN, E., JACKSON, C., AND BONEH, D. An analysis of private browsing modes in modern browsers. In *Proceedings of the 19th USENIX Security Symposium* (2010), pp. 6–6.
- [9] AGTEN, P., VAN ACKER, S., BRONDSEMA, Y., PHUNG, P. H., DESMET, L., AND PIJSESENS, F. JSand: Complete client-side sandboxing of third-party JavaScript without browser modifications. In *Proceedings of the*

- 28th Annual Computer Security Applications Conference (ACSAC) (2012), pp. 1–10.
- [10] AKHawe, D., Barth, A., Lam, P. E., Mitchell, J. C., and Song, D. Towards a formal foundation of web security. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium (CSF)* (2010), pp. 290–304.
  - [11] Akhawe, D., Saxena, P., and Song, D. Privilege separation in HTML5 applications. In *Proceedings of the 21st USENIX Security Symposium* (2012), pp. 429–444.
  - [12] Alcorn, W. Browser Exploitation Framework (BeEF). *Online at <http://beefproject.com>* (2013).
  - [13] Alcorn, W., Frichot, C., and Orru, M. *The browser hacker's handbook*. John Wiley & sons, 2014.
  - [14] Alfardan, N., Bernstein, D. J., Paterson, K. G., Poettering, B., and Schuldt, J. On the security of RC4 in TLS and WPA. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (SP)* (2013).
  - [15] Alfardan, N. J., and Paterson, K. G. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (SP)* (2013).
  - [16] ANDROID DEVELOPERS. Android interface definition language (AIDL). *Online at <http://developer.android.com/guide/components/aidl.html>* (2014).
  - [17] APACHE SOFTWARE FOUNDATION. Apache tomcat - migration guide. *Online at <http://tomcat.apache.org/migration-7.html>* (2013).
  - [18] APPLE. iPhone 5s: About touch ID security. *Online at <http://support.apple.com/kb/HT5949>* (2014).
  - [19] ASSOCIATED PRESS. New nuclear sub is said to have special eavesdropping ability. *Online at [http://www.nytimes.com/2005/02/20/politics/20submarine.html?\\_r=0](http://www.nytimes.com/2005/02/20/politics/20submarine.html?_r=0)* (2005).
  - [20] BAHAJI, Z. A., and ILLYES, G. Https as a ranking signal. *Online at <http://googlewebmastercentral.blogspot.be/2014/08/https-as-ranking-signal.html>* (2014).
  - [21] BANK OF AMERICA. SiteKey Security from Bank of America. *Online at <https://www.bankofamerica.com/privacy/online-mobile-banking-privacy/sitekey.go>*.

- [22] BANSAL, C., BHARGAVAN, K., AND MAFFEIS, S. Discovering concrete attacks on website authorization by formal analysis. In *Proceedings of the 25th IEEE Computer Security Foundations Symposium (CSF)* (2012), pp. 247–262.
- [23] BARTH, A. HTTP state management mechanism. *RFC Proposed Standard (RFC 6256)* (2011).
- [24] BARTH, A. The Web Origin Concept. *RFC 6454* (2011).
- [25] BARTH, A., FELT, A. P., SAXENA, P., AND BOODMAN, A. Protecting browsers from extension vulnerabilities. In *Proceedings of the 17th Annual Network and Distributed System Security Conference (NDSS)* (2010).
- [26] BARTH, A., JACKSON, C., AND MITCHELL, J. C. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)* (2008), pp. 75–88.
- [27] BARTH, A., JACKSON, C., AND MITCHELL, J. C. Securing frame communication in browsers. *Communications of the ACM* 52, 6 (2009), 83–91.
- [28] BARTH, A., VEDITZ, D., AND WEST, M. Content Security Policy Level 2. *W3C Working Draft* (2014).
- [29] BATES, D., BARTH, A., AND JACKSON, C. Regular expressions considered harmful in client-side xss filters. In *Proceedings of the 19th international conference on World wide web (WWW)* (2010), pp. 91–100.
- [30] BBC. Privacy and cookies. *Online at <http://www.bbc.co.uk/privacy/>* (2012).
- [31] BELSHE, M., AND PEON, R. SPDY Protocol. *IETF Internet Draft* (2012).
- [32] BELSHE, M., THOMSON, M., MELNIKOV, A., AND PEON, R. Hypertext Transfer Protocol version 2.0. *IETF Internet Draft* (2014).
- [33] BERG, D. How to use your fingerprint reader. *Online at <http://blog.laptopmag.com/how-to-use-your-fingerprint-reader>* (2012).
- [34] BERGKVIST, A., BURNETT, D. C., JENNINGS, C., AND NARAYANAN, A. WebRTC 1.0: Real-Time Communication Between Browsers. *W3C Working Draft* (2013).
- [35] BERJON, R., FAULKNER, S., LEITCHHEAD, T., NAVARA, E. D., O'CONNOR, E., PFEIFFER, S., AND HICKSON, I. HTML 5.1 Specification. *W3C Working Draft* (2014).

- [36] BERJON, R., FAULKNER, S., LEITHEAD, T., NAVARA, E. D., O'CONNOR, E., PFEIFFER, S., AND HICKSON, I. HTML 5.1 Specification - The sandbox Attribute. *W3C Working Draft* (2014).
- [37] BERNERS-LEE, T., FIELDING, R. T., AND MASINTER, L. Uniform Resource Identifier (URI): Generic Syntax. *RFC Internet Standard (RFC 3986)* (2005).
- [38] BIRGISSON, A., POLITZ, J., ERLINGSSON, Ú., TALY, A., VRABLE, M., AND LENTCZNER, M. Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud. In *Proceedings of the 21st Annual Network and Distributed System Security Conference (NDSS)* (2014).
- [39] BLOCK, S., AND POPESCU, A. DeviceOrientation Event Specification. *W3C Working Draft* (2011).
- [40] BOLIN, M., AND MILLER, R. Chickenfoot for firefox: Rewrite the web. *Online at <http://groups.csail.mit.edu/uid/chickenfoot/>* (2009).
- [41] BORTZ, A., BARTH, A., AND CZEKIS, A. Origin cookies: Session integrity for web applications. *Web 2.0 Security and Privacy (W2SP)* (2011).
- [42] Bos, B. CSS Mobile Profile 2.0. *W3C Working Group Note* (2014).
- [43] BRAUN, B. Deliverable 2.3: Secure interaction specification and enforcement. *Online at <https://www.websand.eu/deliverables/WP2/D2.3.pdf>* (2013).
- [44] BRAY, T. The javascript object notation (JSON) data interchange format. *RFC Proposed Standard (RFC 7159)* (2014).
- [45] BUGLIESI, M., CALZAVARA, S., FOCARDI, R., KHAN, W., AND TEMPESTA, M. Provably sound browser-based enforcement of web session integrity. In *Proceedings of the 27th IEEE Computer Security Foundations Symposium (CSF)* (2014).
- [46] BURNETT, D. C., BERGKVIST, A., JENNINGS, C., AND ANANT, N. Media Capture and Streams. *W3C Working Draft* (2013).
- [47] BURNS, J. Cross site reference forgery: An introduction to a common web application weakness. *Online at [https://www.isecpartners.com/media/11961/csrf\\_paper.pdf](https://www.isecpartners.com/media/11961/csrf_paper.pdf)* (2005).
- [48] BUTLER, E. Firesheep. *Online at <http://codebutler.com/firesheep>* (2010).

- [49] CALZAVARA, S., TOLOMEI, G., BUGLIESI, M., AND ORLANDO, S. Quite a mess in my cookie jar!: leveraging machine learning to protect web authentication. In *Proceedings of the 23rd international conference on World Wide Web (WWW)* (2014), pp. 189–200.
- [50] CARLINI, N., FELT, A. P., AND WAGNER, D. An evaluation of the Google Chrome extension security architecture. In *Proceedings of the 21st USENIX Security Symposium* (2012).
- [51] CARLISLE, D., ION, P., AND MINER, R. Mathematical Markup Language (MathML) Version 3.0 . *W3C Recommendation* (2010).
- [52] CHEN, E. Y., BAU, J., REIS, C., BARTH, A., AND JACKSON, C. App isolation: get the security of multiple browsers with just one. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)* (2011), pp. 227–238.
- [53] CHEN, P., NIKIFORAKIS, N., DESMET, L., AND HUYGENS, C. A dangerous mix: Large-scale analysis of mixed-content websites. In *Proceedings of the 16th Information Security Conference (ISC)* (2013).
- [54] CHIN, E., AND WAGNER, D. Bifocals: Analyzing webview vulnerabilities in android applications. In *Information Security Applications*. Springer, 2014, pp. 138–159.
- [55] CHOU, N., LEDESMA, R., TERAGUCHI, Y., AND MITCHELL, J. C. Client-side defense against web-based identity theft. In *Proceedings of the 11th Annual Network and Distributed System Security Conference (NDSS)* (2004).
- [56] COOPER, D., SANTESSON, S., FARRELL, S., BOEYEN, S., HOUSLEY, R., AND POLK, W. Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile. *RFC Proposed Standard (RFC 5280)* (2008).
- [57] CRAWFORD, D. Why mobile web apps are slow. *Online at <http://sealedabstract.com/rants/why-mobile-web-apps-are-slow/>* (2013).
- [58] Czeskis, A., Moshchuk, A., Kohno, T., and Wang, H. J. Lightweight server support for browser-based CSRF protection. In *Proceedings of the 22nd international conference on World Wide Web (WWW)* (2013), pp. 273–284.
- [59] DACOSTA, I., CHAKRADEO, S., AHAMAD, M., AND TRAYNOR, P. One-time cookies: Preventing session hijacking attacks with stateless

- authentication tokens. *ACM Transactions on Internet Technology (TOIT)* 12, 1 (2012), 31.
- [60] DAHL, D., AND SLEEVÍ, R. Web Cryptography API. *W3C Last Call Working Draft* (2014).
- [61] DAHLSTRÖM, E., DENGLER, P., GRASSO, A., LILLEY, C., MCCORMACK, C., SCHEPERS, D., AND WATT, J. Scalable Vector Graphics (SVG) 1.1 (Second Edition). *W3C Recommendation* (2011).
- [62] DE GROEF, W., DEVRIESE, D., NIKIFORAKIS, N., AND PIJSSSENS, F. Flowfox: a web browser with flexible and precise information flow control. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)* (2012).
- [63] DE RYCK, P., DECAT, M., DESMET, L., PIJSSSENS, F., AND JOOSEN, W. Security of web mashups: a survey. In *Proceedings of the 15th Nordic Conference on Secure IT Systems (NordSec)* (2010), pp. 223–238.
- [64] DE RYCK, P., DESMET, L., HEYMAN, T., PIJSSSENS, F., AND JOOSEN, W. CsFire: Transparent client-side mitigation of malicious cross-domain requests. In *Proceedings of the 2nd International Symposium on Engineering Secure Software and Systems (ESSoS)* (2010), pp. 18–34.
- [65] DE RYCK, P., DESMET, L., JOOSEN, W., AND PIJSSSENS, F. Automatic and precise client-side protection against CSRF attacks. In *Proceedings of the 16th European Symposium on Research in Computer Security (ESORICS)* (2011), pp. 100–116.
- [66] DE RYCK, P., DESMET, L., PHILIPPAERTS, P., AND PIJSSSENS, F. A security analysis of next generation web standards. Tech. rep., European Network and Information Security Agency (ENISA), July 2011.
- [67] DE RYCK, P., DESMET, L., PIJSSSENS, F., AND JOHNS, M. *Primer on Client-side Web Security*. Springer, 2014.
- [68] DE RYCK, P., DESMET, L., PIJSSSENS, F., AND JOOSEN, W. Automatic and precise client-side protection against CSRF attacks - downloads. *Online at <https://distrinet.cs.kuleuven.be/software/CsFire/esorics2011/>* (2011).
- [69] DE RYCK, P., DESMET, L., PIJSSSENS, F., AND JOOSEN, W. A security analysis of emerging web standards – HTML5 and friends, from specification to implementation. In *Proceedings of the 11th International Conference on Security and Cryptography (SECRYPT)* (2012), pp. 257–262.

- [70] DE RYCK, P., DESMET, L., PIJSESENS, F., AND JOOSEN, W. Securing web applications with browser add-ons: an experience report. In *Presented at the Grande Region Security and Reliability Day (GRSRD)* (2013).
- [71] DE RYCK, P., DESMET, L., PIJSESENS, F., AND JOOSEN, W. Eradicating bearer tokens for session management. *W3C/IAB Workshop on Strengthening the Internet Against Pervasive Monitoring (STRINT)* (2014).
- [72] DE RYCK, P., DESMET, L., PIJSESENS, F., AND JOOSEN, W. SecSess: Keeping your session tucked away in your browser. In *Proceedings of the 30th ACM Symposium on Applied Computing (SAC), To be published* (2015).
- [73] DE RYCK, P., NIKIFORAKIS, N., DESMET, L., AND JOOSEN, W. Tabshots: Client-side detection of tabnabbing attacks. In *Proceedings of the 8th ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (2013), pp. 447–456.
- [74] DE RYCK, P., NIKIFORAKIS, N., DESMET, L., PIJSESENS, F., AND JOOSEN, W. Serene: self-reliant client-side protection against session fixation. In *Proceedings of the 12th International IFIP Conference on Distributed Applications and Interoperable Systems (DAIS)* (2012), pp. 59–72.
- [75] DELIA ONLINE. Cookies used on delia online. *Online at <http://www.deliaonline.com/home/delia-online-cookies.html>* (2012).
- [76] DEVELOPERS, G. Safe Browsing API. *Online at <https://developers.google.com/safe-browsing/>* (2014).
- [77] DHAMIJA, R., AND TYGAR, J. D. The battle against phishing: Dynamic security skins. In *Proceedings of the 1st Symposium on Usable Privacy and Security (SOUPS)* (2005), pp. 77–88.
- [78] DHAMIJA, R., TYGAR, J. D., AND HEARST, M. Why phishing works. In *Proceedings of the ACM CHI conference on Human Factors in computing systems (CHI)* (2006), pp. 581–590.
- [79] DIERKS, T. The transport layer security (TLS) protocol version 1.2. *RFC 5246* (2008).
- [80] DIERKS, T., AND RESCORLA, E. The transport layer security (TLS) protocol version 1.3. *RFC 5246bis* (2014).

- [81] DIETZ, M., CZEKIS, A., BALFANZ, D., AND WALLACH, D. S. Origin-bound certificates: A fresh approach to strong client authentication for the web. In *Proceedings of the 21st USENIX Security Symposium* (2012), pp. 16–16.
- [82] DINGLEDINE, R., MATHEWSON, N., AND SYVERSON, P. Tor: The second-generation onion router. Tech. rep., DTIC Document, 2004.
- [83] DJANGO. Cross site request forgery protection. *Online at <http://docs.djangoproject.com/en/dev/ref/contrib/csrf/>* (2011).
- [84] DUBROY, P. How many tabs do people use? (Now with real data!). *Online at <http://dubroy.com/blog/how-many-tabs-do-people-use-now-with-real-data/>* (2009).
- [85] EASTLAKE 3RD, D. Transport layer security (TLS) extensions: Extension definitions. *RFC Proposed Standard (RFC 6066)* (2011).
- [86] ECKERSLEY, P. How unique is your web browser? In *Proceedings of the 10th Privacy Enhancing Technologies Symposium (PETS)*. 2010, pp. 1–18.
- [87] ECKERSLEY, P., AND BURNS, J. The eff ssl observatory. *Online at <https://www.eff.org/observatory>* (2010).
- [88] EGELMAN, S., CRANOR, L. F., AND HONG, J. You've been warned: an empirical study of the effectiveness of web browser phishing warnings. In *Proceedings of the ACM CHI conference on Human Factors in computing systems (CHI)* (2008), pp. 1065–1074.
- [89] ELECTRONIC FRONTIER FOUNDATION. HTTPS everywhere. *Online at <https://www.eff.org/https-everywhere>* (2013).
- [90] EMC. RSA SecurID - Two-Factor Authentication Security Token. *Online at <http://www.emc.com/security/rsa-securid.htm>* (2013).
- [91] ERLINGSSON, Ú. The inlined reference monitor approach to security policy enforcement. Tech. rep., Cornell University, 2003.
- [92] ETEMAD, E. J. Cascading Style Sheets (CSS) Snapshot 2010. *W3C Working Group Note* (2010).
- [93] ETTERCAP PROJECT. Ettercap home page. *Online at <http://ettercap.github.io/ettercap/>* (2013).
- [94] EVANS, C., PALMER, C., AND SLEEV, R. Public Key Pinning Extension for HTTP. *IETF Internet Draft* (2014).

- [95] FACEBOOK. Facebook login. *Online at <http://developers.facebook.com/docs/facebook-login/>* (2013).
- [96] FACEBOOK HELP CENTER. Extra security features. *Online at <https://www.facebook.com/help/413023562082171/>* (2014).
- [97] FARREL, S., AND TSCHOFENIG, H. Pervasive Monitoring is an Attack. *RFC Best Current Practice (RFC 7258)* (2014).
- [98] FERGAL GLYNN, V. Static code analysis. *Online at <http://www.veracode.com/security/static-code-analysis>* (2013).
- [99] FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MASINTER, L., LEACH, P., AND BERNERS-LEE, T. Hypertext Transfer Protocol – HTTP/1.1. *RFC 2616* (1999).
- [100] FITZGERALD, D. Yahoo passwords stolen in latest data breach. *Online at <http://online.wsj.com/news/articles/SB10001424052702304373804577522613740363638>* (2012).
- [101] FLORÊNCIO, D., AND HERLEY, C. Sex, lies and cyber-crime surveys. In *Economics of Information Security and Privacy III*. Springer, 2013, pp. 35–53.
- [102] FORUMS, I. Which is the best way to configure ABE? *Online at <http://forums.informaction.com/viewtopic.php?f=23&t=4752>* (July 2010).
- [103] FRIEDL, S., AND POPOV, A. Transport Layer Security (TLS) Application Layer Protocol Negotiation Extension. *RFC Proposed Standard (RFC 7301)* (2014).
- [104] FUNG, B. S., AND LEE, P. P. A privacy-preserving defense mechanism against request forgery attacks. In *Proceedings of the 10th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)* (2011), pp. 45–52.
- [105] GEIER, E. Prevent wi-fi eavesdroppers from hijacking your accounts. *Online at <http://www.ciscopress.com/articles/article.asp?p=1750204>* (2011).
- [106] GOLLUCCI, P. M. Apache.org incident report for 04/09/2010. *Online at [https://blogs.apache.org/infra/entry/apache\\_org\\_04\\_09\\_2010](https://blogs.apache.org/infra/entry/apache_org_04_09_2010)* (2010).
- [107] GOOGLE. Trusted computers. *Online at <https://support.google.com/accounts/answer/2544838?hl=en>* (2014).

- [108] GRANT, A. C. Search for trust: An analysis and comparison of CA system alternatives and enhancements.
- [109] GUARNIERI, S., AND LIVSHITS, V. B. GATEKEEPER: Mostly static enforcement of security and reliability policies for javascript code. In *Proceedings of the 18th USENIX Security Symposium* (2009), pp. 151–168.
- [110] HADDIX, J., AND MIESSLER, D. OWASP top 10 mobile risks. *Online at [https://www.owasp.org/index.php/Projects/OWASP\\_Mobile\\_Security\\_Project\\_-\\_Top\\_Ten\\_Mobile\\_Risks](https://www.owasp.org/index.php/Projects/OWASP_Mobile_Security_Project_-_Top_Ten_Mobile_Risks)* (2013).
- [111] HAK5. Wifi pineapple. *Online at <https://wifipineapple.com/>* (2013).
- [112] HALLAM-BAKER, P. Http integrity header. *IETF Internet Draft* (2012).
- [113] HALLGREN, P. A., MAURITZSON, D. T., AND SABELFELD, A. Glasstube: a lightweight approach to web application integrity. In *Proceedings of the 8th ACM SIGPLAN workshop on Programming languages and analysis for security (PLAS)* (2013), pp. 71–82.
- [114] HEPPER, D. Gmail CSRF vulnerability explained. *Online at <http://daniel.hepper.net/blog/2008/11/gmail-csrf-vulnerability-explained/>* (2008).
- [115] HICKSON, I. HTML5 Web Messaging. *W3C Candidate Recommendation* (2012).
- [116] HICKSON, I. Server-Sent Events. *W3C Candidate Recommendation* (2012).
- [117] HICKSON, I. Web Workers. *W3C Candidate Recommendation* (2012).
- [118] HICKSON, I. Web Storage. *W3C Recommendation* (2013).
- [119] HIROSHIMA, N. How I lost my \$50,000 twitter username. *Online at <https://medium.com/@N/how-i-lost-my-50-000-twitter-username-24eb09e026dd>* (2014).
- [120] HIRSCH, F. Device APIs Working Group. *Online at <http://www.w3.org/2009/dap/>* (2014).
- [121] HODGES, J., JACKSON, C., AND BARTH, A. HTTP strict transport security (HSTS). *RFC Proposed Standard (RFC 6797)* (2012).
- [122] HOFFMAN, P., AND SCHLYTER, J. The DNS-based authentication of named entities (DANE) transport layer security (TLS) protocol: TLSA. *RFC Proposed Standard (RFC 6698)* (2012).

- [123] HONAN, M. How Apple and Amazon security flaws led to my epic hacking. *Online at <http://www.wired.com/2012/08/apple-amazon-mat-honan-hacking/>* (2012).
- [124] HP ENTERPRISE SECURITY. HP Fortify Static Code Analyzer (SCA). *Online at <http://www.hpenterprisesecurity.com/products/hp-fortify-software-security-center/hp-fortify-static-code-analyzer>* (2013).
- [125] HUANG, L.-S., AND JACKSON, C. Clickjacking attacks unresolved. *Online at [https://docs.google.com/document/pub?id=1hVcxPeCidZrM5acFH9ZoTYzg1D0VjkG3BDW\\_oUdn5qc](https://docs.google.com/document/pub?id=1hVcxPeCidZrM5acFH9ZoTYzg1D0VjkG3BDW_oUdn5qc)* (2011).
- [126] HUANG, L.-S., MOSHCHUK, A., WANG, H. J., SCHECHTER, S., AND JACKSON, C. Clickjacking: attacks and defenses. In *Proceedings of the 21st USENIX Security Symposium* (2012), pp. 22–22.
- [127] HUANG, L.-S., RICE, A., ELLINGSEN, E., AND JACKSON, C. Analyzing forged SSL certificates in the wild. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (SP)* (2014).
- [128] HUGHES, E. An encrypted key transmission protocol. *rump session of CRYPTO 94* (1994).
- [129] IANA. Resource identifier scheme name: facetime. *Online at <https://www.iana.org/assignments/uri-schemes/prov/facetime>* (2012).
- [130] ICHNOWSKI, J., AND MANICO, J. OWASP’s java xml templates. *Online at <http://code.google.com/p/owasp-jxt/>* (2013).
- [131] ICHNOWSKI, J., MANICO, J., AND LONG, J. OWASP java encoder project. *Online at [https://www.owasp.org/index.php/OWASP\\_Java\\_Encoder\\_Project](https://www.owasp.org/index.php/OWASP_Java_Encoder_Project)* (2013).
- [132] INFOSECURITY. Adobe hacked – customers’ card details and adobe source code stolen. *Online at <http://www.infosecurity-magazine.com/view/34872/adobe-hacked-customers-card-details-and-adobe-source-code-stolen>* (2013).
- [133] INFOSECURITY. How GCHQ hacked belgacom. *Online at <http://www.infosecurity-magazine.com/view/35558/how-gchq-hacked-belgacom>* (2013).
- [134] INFOSECURITY. 360 million stolen credentials and 1.25 billion email addresses found on the black market. *Online at <http://www.infosecurity-magazine.com/view/37135/360->*

*million-stolen-credentials-and-125-billion-email-addresses-found-on-the-black-market/* (2014).

- [135] INTERNET EXPLORER DEVELOPER CENTER. Making HTML safer: details for toStaticHTML (Windows Store apps using JavaScript and HTML). *Online at <http://msdn.microsoft.com/en-us/library/ie/hh465388.aspx>* (2012).
- [136] JACKSON, C., AND BARTH, A. Beware of finer-grained origins. In *Web 2.0 Security and Privacy (W2SP)* (2008).
- [137] JACKSON, C., AND BARTH, A. ForceHTTPS: protecting high-security web sites from network attacks. In *Proceedings of the 17th international conference on World Wide Web (WWW)* (2008), pp. 525–534.
- [138] JACOBS, F. How Reuters got compromised by the Syrian electronic army. *Online at <https://medium.com/@FredericJacobs/the-reuters-compromise-by-the-syrian-electronic-army-6bf570e1a85b>* (2014).
- [139] JANG, D., JHALA, R., LERNER, S., AND SHACHAM, H. An empirical study of privacy-violating information flows in javascript web applications. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)* (2010), pp. 270–283.
- [140] JOHNS, M. Sessionsafe: Implementing xss immune session handling. In *Proceedings of the 11th European Symposium on Research in Computer Security (ESORICS)* (2006), pp. 444–460.
- [141] JOHNS, M., BRAUN, B., SCHRANK, M., AND POSEGGA, J. Reliable protection against session fixation attacks. In *Proceedings of the 26th ACM Symposium on Applied Computing (SAC)* (2011), pp. 1531–1537.
- [142] JOHNS, M., LEKIES, S., BRAUN, B., AND FLESCH, B. Betterauth: Web authentication revisited. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)* (2012), pp. 169–178.
- [143] JOHNS, M., AND WINTER, J. Requestrodeo: Client side protection against session riding. In *Proceedings of the OWASP AppSec Europe 2006 Conference (AppSecEU)* (2006), pp. 5–17.
- [144] JOVANOVIC, N., KIRDA, E., AND KRUEGEL, C. Preventing cross site request forgery attacks. *Proceedings of the 2nd International Conference on Security and Privacy in Communication Networks (SecureComm)* (2006), 1–10.
- [145] KELION, L. eBay redirect attack puts buyers' credentials at risk. *Online at <http://www.bbc.com/news/technology-29241563>* (2014).

- [146] KELLY, S. M. LastPass passwords exposed for some internet explorer users. *Online at <http://mashable.com/2013/08/19/lastpass-password-bug/>* (2013).
- [147] KING, A. Club nintendo japan hacked, user details could be compromised. *Online at <http://wiiudaily.com/2013/07/club-nintendo-japan-hacked/>* (2013).
- [148] KIRDA, E., KRUEGEL, C., VIGNA, G., AND JOVANOVIC, N. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *Proceedings of the 21st ACM Symposium on Applied Computing (SAC)* (2006), pp. 330–337.
- [149] KOVACS, E. CSRF vulnerability in eBay allows hackers to hijack user accounts. *Online at <http://news.softpedia.com/news/CSRF-Vulnerability-in-eBay-Allows-Hackers-to-Hijack-User-Accounts-Video-383316.shtml>* (2013).
- [150] KOVACS, E. Vodafone germany hacked, details of 2 million users stolen. *Online at <http://news.softpedia.com/news/Vodafone-Germany-Hacked-Details-of-2-Million-Users-Stolen-382458.shtml>* (2013).
- [151] LAMOURI, M. The Network Information API. *W3C Working Draft* (2012).
- [152] LANGBERG, M. AOL acts to thwart hackers. *Online at [http://simson.net/clips/1995/95.SJMN.AOL\\_Hackers.html](http://simson.net/clips/1995/95.SJMN.AOL_Hackers.html)* (1995).
- [153] Langley, A. Overclocking ssl. *Online at <https://www.imperialviolet.org/2010/06/25/overclocking-ssl.html>* (2010).
- [154] Langley, A. Revocation checking and chrome's crl. *Online at <https://www.imperialviolet.org/2012/02/05/crlsets.html>* (2012).
- [155] Langley, A. ChaCha20 and Poly1305 based Cipher Suites for TLS. *IETF Internet Draft* (2013).
- [156] LAURIE, B., LANGLEY, A., AND KASPER, E. Certificate transparency. *RFC Experimental (RFC 6962)* (2013).
- [157] LEKIES, S., HEIDERICH, M., APPELT, D., HOLZ, T., AND JOHNS, M. On the fragility and limitations of current browser-provided clickjacking protection schemes. In *Proceedings of the 6th USENIX Workshop on Offensive technologies (WOOT)* (2012), pp. 53–63.

- [158] LEKIES, S., STOCK, B., AND JOHNS, M. 25 million flows later: large-scale detection of DOM-based XSS. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)* (2013), pp. 1193–1204.
- [159] LEKIES, S., TIGHZERT, W., AND JOHNS, M. Towards stateless, client-side driven cross-site request forgery protection for web applications. In *Proceedings of the 7th conference on Sicherheit, Schutz und Zuverlässigkeit (Sicherheit)* (2012), pp. 111–121.
- [160] LEYDEN, J. Hackers break onto White House military network. *Online at [http://www.theregister.co.uk/2012/10/01/white\\_house\\_hack/](http://www.theregister.co.uk/2012/10/01/white_house_hack/)* (2012).
- [161] LINHART, C., KLEIN, A., HELED, R., AND ORRIN, S. Http request smuggling. *Computer Security Journal* 22, 1 (2006), 13.
- [162] LUO, T., HAO, H., DU, W., WANG, Y., AND YIN, H. Attacks on webview in the android system. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC)* (2011), pp. 343–352.
- [163] MAES, W., HEYMAN, T., DESMET, L., AND JOOSEN, W. Brower protection against cross-site request forgery. In *Proceedings of the 1st ACM workshop on Secure execution of untrusted code (SecuCode)* (2009), pp. 3–10.
- [164] MAHEMOFF, M. Explaining the “don’t click” clickjacking tweetbomb. *Online at <http://softwareas.com/explaining-the-dont-click-clickjacking-tweetbomb/>* (2009).
- [165] MAO, Z., LI, N., AND MOLLOY, I. Defeating cross-site request forgery attacks with browser-enforced authenticity protection. *Proceedings of the 13th International Conference on Financial Cryptography and Data Security (FC)* (2009), 238–255.
- [166] MAONE, G. Noscript 2.0.9.9. *Online at <http://noscript.net/>* (2011).
- [167] MAONE, G. NoScript - javascript/java/flash blocker for a safer firefox experience! *Online at <http://noscript.net/>* (2014).
- [168] MAONE, G. NoScript Application Boundaries Enforcer (ABE). *Online at <http://noscript.net/abe/>* (2014).
- [169] MAONE, G., HUANG, D. L.-S., GONDROM, T., AND HILL, B. User Interface Safety Directives for Content Security Policy. *W3C Last Call Working Draft* (2014).

- [170] MARLINSPIKE, M. New tricks for defeating ssl in practice. *BlackHat DC, February* (2009).
- [171] MARLINSPIKE, M. Sslstrip. *Online at <http://www.thoughtcrime.org/software/sslstrip/>* (2009).
- [172] MARTIN, B., BROWN, M., PALLER, A., AND KIRBY, D. Cwe/sans top 25 most dangerous programming errors. *Online at <http://cwe.mitre.org/top25/>* (2011).
- [173] MASINTER, L. The "data" url scheme. *RFC Proposed Standard (RFC 2397)* (1998).
- [174] MASNICK, M. FLYING PIG: The NSA Is Running Man In The Middle Attacks Imitating Google's Servers. *Online at <http://www.techdirt.com/articles/20130910/10470024468/flying-pig-nsa-is-running-man-middle-attacks-imitating-googles-servers.shtml>* (2013).
- [175] MAYER, J., AND NARAYANAN, A. Do not track - universal web tracking opt out. *Online at <http://donottrack.us/>* (2011).
- [176] MEHTA, N., SICKING, J., GRAFF, E., POPESCU, A., ORLOW, J., AND BELL, J. Indexed Database API. *W3C Candidate Recommendation* (2013).
- [177] MICKENS, J. Pivot: Fast, synchronous mashup isolation using generator chains. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (SP)* (2014), pp. 261–275.
- [178] MICROSOFT CORPORATION. Tracking protection lists. *Online at <http://ie.microsoft.com/testdrive/Browser/TrackingProtectionLists/>* (2011).
- [179] MODELL, M., BARZ, A., TOTH, G., AND LOESCH, C. v. Certificate patrol. *Online at <https://addons.mozilla.org/en-US/firefox/addon/certificate-patrol/>* (2014).
- [180] MOZILLA. Jetpack. *Online at <https://wiki.mozilla.org/Jetpack>* (2014).
- [181] MURDOCH, S. J. Hardened stateless session cookies. In *Security Protocols XVI*. 2011, pp. 93–101.
- [182] NIKIFORAKIS, N., INVERNIZZI, L., KAPRAVELOS, A., VAN ACKER, S., JOOSEN, W., KRUEGEL, C., PISSSENS, F., AND VIGNA, G. You are what you include: Large-scale evaluation of remote javascript

- inclusions. In *Proceedings of the 19th ACM Conference on Computer and Communications Security* (2012), pp. 736–747.
- [183] NIKIFORAKIS, N., KAPRAVELOS, A., JOOSEN, W., KRUEGEL, C., PIESSENS, F., AND VIGNA, G. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (SP)* (2013).
- [184] NIKIFORAKIS, N., MAKRIDAKIS, A., ATHANASOPOULOS, E., AND MARKATOS, E. P. Alice, what did you do last time? fighting phishing using past activity tests. In *Proceedings of the 3rd European Conference on Computer Network Defense (EC2ND)* (2009), pp. 107–117.
- [185] NIKIFORAKIS, N., MEERT, W., YOUNAN, Y., JOHNS, M., AND JOOSEN, W. Sessionshield: lightweight protection against session hijacking. In *Proceedings of the 3rd International Symposium on Engineering Secure Software and Systems (ESSoS)* (2011), pp. 87–100.
- [186] NIKIFORAKIS, N., VAN ACKER, S., PIESSENS, F., AND JOOSEN, W. Exploring the ecosystem of referrer-anonymizing services. In *Proceedings of the 12th Privacy Enhancing Technologies Symposium (PETS)* (2012), pp. 259–278.
- [187] NIKIFORAKIS, N., YOUNAN, Y., AND JOOSEN, W. Hproxy: Client-side detection of SSL stripping attacks. In *Proceedings of the 7th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 2010, pp. 200–218.
- [188] NOTTINGHAM, M. Opportunistic encryption for HTTP URIs. *IETF Internet Draft* (2014).
- [189] NOTTINGHAM, M. Opportunistic encryption for HTTP URIs. *IETF Internet Draft* (2014).
- [190] OPENDNS. PhishTank. *Online at <http://www.phishtank.com/>* (2014).
- [191] O'TOOLE, J. Mobile apps overtake PC Internet usage in U.S. *Online at <http://money.cnn.com/2014/02/28/technology/mobile/mobile-apps-internet/>* (2014).
- [192] PELIZZI, R., AND SEKAR, R. A server-and browser-transparent CSRF defense for web 2.0 applications. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC)* (2011), pp. 257–266.

- [193] PERLROTH, N. Lax security at linkedin is laid bare. *Online at <http://www.nytimes.com/2012/06/11/technology/linkedin-breach-exposes-light-security-even-at-data-companies.html?pagewanted=all>* (2012).
- [194] PHUNG, P. H., SANDS, D., AND CHUDNOV, A. Lightweight self-protecting javascript. In *Proceedings of the 4th ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (2009), pp. 47–60.
- [195] POEPLAU, S., FRATANTONIO, Y., BIANCHI, A., KRUEGEL, C., AND VIGNA, G. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *Proceedings of the 21st Annual Network and Distributed System Security Conference (NDSS)* (2014), pp. 23–26.
- [196] POPESCU, A. Geolocation API Specification. *W3C Recommendation* (2013).
- [197] PRINS, J. Diginotar certificate authority breach – ‘operation black tulip’. *Fox-IT* (2011).
- [198] QUALYS. Trustworthy internet movement - SSL pulse. *Online at <https://www.trustwothyinternet.org/ssl-pulse/>* (2014).
- [199] RANGANATHAN, A., AND SICKING, J. File API. *W3C Last Call Working Draft* (2013).
- [200] RAPID7. Metasploit. *Online at <http://www.metasploit.com/>* (2013).
- [201] RASKIN, A. Tabnabbing: A new type of phishing attack. *Online at <http://www.azarask.in/blog/post/a-new-type-of-phishing-attack/>* (2010).
- [202] REISINGER, D. eBay hacked, requests all users change passwords. *Online at <http://www.cnet.com/news/ebay-hacked-requests-all-users-change-passwords/>* (2014).
- [203] RILEY, M., ELGIN, B., LAWRENCE, D., AND MATLACK, C. Missed alarms and 40 million stolen credit card numbers: How target blew it. *Online at <http://www.businessweek.com/articles/2014-03-13/target-missed-alarms-in-epic-hack-of-credit-card-data>* (2014).
- [204] ROBERTS, P. F. 7 ways to beat fingerprint biometrics. *Online at <http://www.itworld.com/slideshow/120606/7-ways-beat-fingerprint-biometrics-374041>* (2013).

- [205] Ross, D. IE 8 XSS filter architecture / implementation. *Online at <http://blogs.technet.com/b/srd/archive/2008/08/19/ie-8-xss-filter-architecture-implementation.aspx>* (2008).
- [206] Ross, D. Entry point regulation for web apps. *Online at <http://randomdross.blogspot.be/2014/08/entry-point-regulation-for-web-apps.html>* (2014).
- [207] Ross, D., AND GONDROM, T. HTTP header field X-Frame-Options. *RFC Informational (RFC 7034)* (2013).
- [208] RUBY ON RAILS. Actioncontroller::requestforgeryprotection. *Online at <http://api.rubyonrails.org/classes/ActionController/RequestForgeryProtection.html>* (2011).
- [209] RYDSTEDT, G., BURSZTEIN, E., BONEH, D., AND JACKSON, C. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. *Web 2.0 Security and Privacy (W2SP)* (2010).
- [210] RYDSTEDT, G., GOURDIN, B., BURSZTEIN, E., AND BONEH, D. Framing attacks on smart phones and dumb routers: tap-jacking and geo-localization attacks. In *Proceedings of the 4th USENIX Workshop on Offensive technologies (WOOT)* (2010), pp. 1–8.
- [211] SAMUEL, J. Requestpolicy 0.5.20. *Online at <http://www.requestpolicy.com>* (2011).
- [212] SAMUEL, J., AND ZHANG, B. Requestpolicy: Increasing web browsing privacy through control of cross-site requests. In *Proceedings of the 9th Privacy Enhancing Technologies Symposium (PETS)* (2009), pp. 128–142.
- [213] SAMUEL, M., SAXENA, P., AND SONG, D. Context-sensitive auto-sanitization in web templating languages using type qualifiers. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)* (2011), pp. 587–600.
- [214] SANDLER, D. R., AND WALLACH, D. S. <input type=“password”>must die! In *Web 2.0 Security and Privacy (W2SP)* (2008).
- [215] SAXENA, P., AKHawe, D., HANNA, S., MAO, F., MCCAMANT, S., AND SONG, D. A symbolic execution framework for javascript. In *Proceedings of the 31st IEEE Symposium on Security and Privacy (SP)* (2010), pp. 513–528.
- [216] SAXENA, P., MOLNAR, D., AND LIVSHITS, B. SCRIPTGARD: automatic context-sensitive sanitization for large-scale legacy web

- applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)* (2011), pp. 601–614.
- [217] SCHNEIER, B. *Applied cryptography: protocols, algorithms, and source code in C*. John Wiley & sons, 2007.
- [218] SCHOEN, S., AND GALPERIN, E. Iranian man-in-the-middle attack against google demonstrates dangerous weakness of certificate authorities. *Online at <https://www.eff.org/deeplinks/2011/08/iranian-man-middle-attack-against-google>* (2011).
- [219] SCHRANK, M., BRAUN, B., JOHNS, M., AND POSEGGA, J. Session fixation - the forgotten vulnerability? In *Proceedings of the 5th conference on Sicherheit, Schutz und Zuverlässigkeit (Sicherheit)* (2010).
- [220] SHAHRIAR, H., AND ZULKERNINE, M. Client-side detection of cross-site request forgery attacks. In *Proceedings of the 21st International Symposium on Software Reliability Engineering (ISSRE)* (2010), pp. 358–367.
- [221] SHEFFER, Y., HOLZ, R., AND SAINT-ANDRE, P. Recommendations for Secure Use of TLS and DTLS. *IETF Internet Draft* (2014).
- [222] SILES, R. Session management cheat sheet - renew the session ID after any privilege level change. *Online at [https://www\\_OWASP\\_org\\_index.php/Session\\_Management\\_Cheat\\_Sheet#Renew\\_the\\_Session\\_ID\\_After\\_Any\\_Privilege\\_Level\\_Change](https://www_OWASP_org_index.php/Session_Management_Cheat_Sheet#Renew_the_Session_ID_After_Any_Privilege_Level_Change)* (2013).
- [223] SINGH, K., MOSHCHUK, A., WANG, H. J., AND LEE, W. On the incoherencies in web browser access control policies. In *Proceedings of the 31st IEEE Symposium on Security and Privacy (SP)* (2010), pp. 463–478.
- [224] SONG, D. dsniff. *Online at <http://www.monkey.org/~dugsong/dsniff/>* (2000).
- [225] SQUID PROJECT MAINTAINERS. squid: Optimising Web Delivery. *Online at <http://www.squid-cache.org/>* (2014).
- [226] STALLINGS, W. *Handbook of computer-communications standards; Vol. 1: the open systems interconnection (OSI) model and OSI-related standards*. Macmillan Publishing Co., Inc., 1987.
- [227] STAMM, S., STERNE, B., AND MARKHAM, G. Reining in the web with content security policy. In *Proceedings of the 19th international conference on World wide web (WWW)* (2010), pp. 921–930.

- [228] STATCOUNTER. Screen resolution alert for web developers. *Online at <http://gs.statcounter.com/press/screen-resolution-alert-for-web-developers>* (2012).
- [229] STERNE, B., AND BARTH, A. Content Security Policy 1.0. *W3C Candidate Recommendation* (2012).
- [230] STOCK, B., LEKIES, S., MUELLER, T., SPIEGEL, P., AND JOHNS, M. Precise client-side protection against DOM-based cross-site scripting. In *Proceedings of the 23rd USENIX Security Symposium* (2014), pp. 655–670.
- [231] STONE, P. Next generation clickjacking. *BlackHat Europe* (2010).
- [232] SURI, R. K., TOMAR, D. S., AND SAHU, D. R. An approach to perceive tabnabbing attack. *International Journal of Scientific & Technology Research 1* (2012).
- [233] SYMANTEC CORPORATION. 2013 Norton report. *Online at [http://www.symantec.com/about/news/resources/press\\_kits/detail.jsp?pkid=norton-report-2013](http://www.symantec.com/about/news/resources/press_kits/detail.jsp?pkid=norton-report-2013)* (2013).
- [234] SYMANTEC CORPORATION. Internet security threat report. *Online at [http://www.symantec.com/content/en/us/enterprise/other\\_resources/b-istr\\_main\\_report\\_v19\\_21291018.en-us.pdf](http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_v19_21291018.en-us.pdf)* (2014).
- [235] TANG, S., DAUTENHAHN, N., AND KING, S. T. Fortifying web-based applications automatically. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CSS)* (2011), pp. 615–626.
- [236] TECHNOESIS. 4 ways to prevent duplicate form submission. *Online at <http://technoesis.net/prevent-double-form-submission/>* (2013).
- [237] TER LOUW, M., GANESH, K. T., AND VENKATAKRISHNAN, V. AdJail: Practical enforcement of confidentiality and integrity policies on web advertisements. In *Proceedings of the 19th USENIX Security Symposium* (2010), pp. 371–388.
- [238] THE APACHE SOFTWARE FOUNDATION. Apache Traffic Server. *Online at <http://trafficserver.apache.org/>* (2014).
- [239] THE GUARDIAN. Edward Snowden. *Online at <http://www.theguardian.com/world/edward-snowden>* (2013).
- [240] THE H SECURITY. Trustwave issued a man-in-the-middle certificate. *Online at <http://h-online.com/-1429982>* (2012).

- [241] THE OWASP FOUNDATION. CSRF guard. *Online at [http://www.owasp.org/index.php/CSRF\\_Guard](http://www.owasp.org/index.php/CSRF_Guard)* (October 2008).
- [242] TNS OPINION & SOCIAL. Special eurobarometer 404 – cyber security. *Online at [http://ec.europa.eu/public\\_opinion/archives/ebs/ebs\\_404\\_en.pdf](http://ec.europa.eu/public_opinion/archives/ebs/ebs_404_en.pdf)* (November 2013).
- [243] TOUSSAIN, M., AND SHIELDS, C. Subterfuge. *Online at <http://kinozoa.com/blog/subterfuge-documentation/>* (2013).
- [244] UNLU, S., AND BICAKCI, K. Notabnab: Protection against the “tabnabbing attack”. In *eCrime Researchers Summit (eCrime)* (2010), pp. 1–5.
- [245] VAN ACKER, S. *Isolating and Restricting Client-Side JavaScript*. PhD thesis, January 2015.
- [246] VAN ACKER, S., DE RYCK, P., DESMET, L., PIJSESENS, F., AND JOOSEN, W. WebJail: least-privilege integration of third-party components in web mashups. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC)* (2011), pp. 307–316.
- [247] VAN ACKER, S., NIKIFORAKIS, N., DESMET, L., PIJSESENS, F., AND JOOSEN, W. Monkey-in-the-browser: Malware and vulnerabilities in augmented browsing script markets. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (2014), ACM, pp. 525–530.
- [248] VAN KESTEREN, A. Cross-Origin Resource Sharing. *W3C Recommendation* (2014).
- [249] VAN KESTEREN, A., AUBOURG, J., SONG, J., AND STEEN, H. R. M. XMLHttpRequest. *W3C Working Draft* (2014).
- [250] VAN KESTEREN, A., GREGOR, A., HUNT, L., AND MS2GER. DOM4. *W3C Working Draft* (2012).
- [251] W3TECHS. Usage statistics and market share of SSL certificate authorities for websites, august 2014. *Online at [http://w3techs.com/technologies/overview/ssl\\_certificate/all](http://w3techs.com/technologies/overview/ssl_certificate/all)* (2014).
- [252] WANG, R., XING, L., WANG, X., AND CHEN, S. Unauthorized origin crossing on mobile platforms: Threats and mitigation. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)* (2013), pp. 635–646.

- [253] WEINBERG, Z., CHEN, E. Y., JAYARAMAN, P. R., AND JACKSON, C. I still know what you visited last summer: Leaking browsing history via user interaction and side channel attacks. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy (SP)* (2011), pp. 147–161.
- [254] WEINBERGER, J., BARTH, A., AND SONG, D. Towards client-side html security policies. In *Proceedings of the 6th USENIX Workshop on Hot Topics on Security (HotSec)* (2011).
- [255] WENYIN, L., HUANG, G., XIAOYUE, L., MIN, Z., AND DENG, X. Detection of phishing webpages based on visual similarity. In *Special interest tracks and posters of the 14th international conference on World Wide Web (WWW)* (2005), pp. 1060–1061.
- [256] WG802.11 - WIRELESS LAN WORKING GROUP. Wireless LAN medium access control (MAC) and physical layer (PHY) specifications: Amendment 6: Medium access control (MAC) security enhancements. *IEEE Standard* (2004).
- [257] WICHERS, D. OWASP top 10. *Online at [https://www.owasp.org/index.php/Category:OWASP\\_Top\\_Ten\\_Project](https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project)* (2013).
- [258] WILLIAMS, J., AND WICHERS, D. OWASP top 10. *Online at [https://www.owasp.org/index.php/Top\\_10\\_2010-Main](https://www.owasp.org/index.php/Top_10_2010-Main)* (2010).
- [259] WU, M., MILLER, R. C., AND GARFINKEL, S. L. Do security toolbars actually prevent phishing attacks? In *Proceedings of the ACM CHI conference on Human Factors in computing systems (CHI)* (2006), pp. 601–610.
- [260] XSSED. XSS Archive. *Online at <http://www.xssed.com/archive/>* (2014).
- [261] YANG, E. Z. HTML Purifier. *Online at <http://htmlpurifier.org/>* (2013).
- [262] ZALEWSKI, M. Arbitrary page mashups (ui redressing). *Online at [http://code.google.com/p/browsersec/wiki/Part2#Arbitrary\\_page\\_mashups\\_\(UI\\_redressing\)](http://code.google.com/p/browsersec/wiki/Part2#Arbitrary_page_mashups_(UI_redressing))* (2010).
- [263] ZALEWSKI, M. Browser security handbook. *Online at <http://code.google.com/p/browsersec/wiki/Main>* (2010).
- [264] ZELLER, W., AND FELTEN, E. W. Cross-site request forgeries: Exploitation and prevention. Tech. rep., Princeton University, 2008.

- [265] ZHANG, Y., HONG, J. I., AND CRANOR, L. F. Cantina: a content-based approach to detecting phishing web sites. In *Proceedings of the 16th international conference on World Wide Web (WWW)* (2007), pp. 639–648.
- [266] ZHOU, Y., AND EVANS, D. Why aren’t http-only cookies more widely deployed? In *Web 2.0 Security and Privacy (W2SP)* (2010).



# List of Publications

## Books

- 2014 DE RYCK, P., DESMET, L., PIJSESENS, F., AND JOHNS, M. *Primer on Client-side Web Security*. Springer, 2014.

## International Journals and Magazines

- 2014 DE RYCK, P., NIKIFORAKIS, N., DESMET, L., PIJSESENS, F., AND JOOSEN, W. Protected Web Components: Hiding Sensitive Content in the Shadows (Tentative title, accepted for publication in IEEE IT Professional).

## Peer-reviewed Papers at International Conferences

- 2015 DE RYCK, P., DESMET, L., PIJSESENS, F., AND JOOSEN, W. SecSess: Keeping your Session Tucked Away in your Browser. In *Proceedings of the 30th ACM Symposium on Applied Computing (SAC)* (2015), To be published.
- 2013 DE RYCK, P., NIKIFORAKIS, N., DESMET, L., AND JOOSEN, W. Tabshots: Client-side detection of tabnabbing attacks. In *Proceedings of the 8th ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (2013), pp. 447–456.
- 2012 DE RYCK, P., DESMET, L., PIJSESENS, F., AND JOOSEN, W. A security analysis of emerging Web standards - HTML5 and friends, from specification to implementation. In *Proceedings of the International*

- Conference on Security and Cryptography (SECRYPT) (2012), pp. 257–262.*
- 2012** DE RYCK, P., NIKIFORAKIS, N., DESMET, L., PIJSSSENS, F., AND JOOSEN, W. Serene: self-reliant client-side protection against session fixation. In *Proceedings of the 12th International IFIP Conference on Distributed Applications and Interoperable Systems (DAIS) (2012)*, pp. 59–72.
- 2011** VAN ACKER, S., DE RYCK, P., DESMET, L., PIJSSSENS, F., AND JOOSEN, W. WebJail: least-privilege integration of third-party components in Web mashups. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC) (2011)*, pp. 307–316.
- 2011** DE RYCK, P., DESMET, L., JOOSEN, W., AND PIJSSSENS, F. Automatic and precise client-side protection against csrf attacks. In *Proceedings of the 16th European Symposium on Research in Computer Security (ESORICS) (2011)*, pp. 100–116.
- 2011** DE RYCK, P., DESMET, L., AND JOOSEN, W. Middleware support for complex and distributed security services in multi-tier Web applications. In *Proceedings of the 3rd International Symposium on Engineering Secure Software and Systems (ESSoS) (2011)*, pp. 114–127.
- 2010** DE RYCK, P., DECAT, M., DESMET, L., PIJSSSENS, F., AND JOOSEN, W. Security of Web mashups: a survey. In *Proceedings of the 15th Nordic Conference on Secure IT Systems (NordSec) (2010)*, pp. 223–238.
- 2010** DE RYCK, P., DESMET, L., HEYMAN, T., PIJSSSENS, F., AND JOOSEN, W. CsFire: Transparent client-side mitigation of malicious cross-domain requests. In *Proceedings of the 2nd International Symposium on Engineering Secure Software and Systems (ESSoS) (2010)*, pp. 18–34.

## Technical Reports

- 2012** DE RYCK, P., DESMET, L., PIJSSSENS, F., AND JOOSEN, W. A security analysis of emerging Web standards - Extended version. Tech. rep., volume CW622, Department of Computer Science, KU Leuven, May 2012.
- 2011** DE RYCK, P., DESMET, L., PHILIPPAERTS, P., AND PIJSSSENS, F. A security analysis of next generation Web standards. Tech. rep., European Network and Information Security Agency (ENISA), July 2011.

## Trainings and Presentations

- 2014 PRENEEL, B., DE RYCK, P., AND NIKIFORAKIS, N. Security Principles for Software Engineering. *Training course at European Space Agency/European Space Operations Centre*
- 2014 DE RYCK, P., MAERIEN, J. AND DESMET, L. Web Security Training. *B-CCENTRE Training day*
- 2014 DE RYCK, P. The Web's security model. *Training session at SecAppDev*
- 2013 DE RYCK, P., DESMET, L., PIJSESENS, F., AND JOOSEN, W. Improving the security of session management in Web applications. *Presented at OWASP AppSec EU (AppSecEU) (2013).*
- 2013 DE RYCK, P., DESMET, L., PIJSESENS, F., AND JOOSEN, W. Securing Web applications with browser add-ons: an experience report. *Presented at the Grande Region Security and Reliability Day (GRSRD) (2013).*
- 2012 DE RYCK, P. HTML 5 Security. *Training session at SecAppDev*
- 2010 DECAT, M., DE RYCK, P., DESMET, L., PIJSESENS, F., AND JOOSEN, W. Towards building secure Web mashups. *Presented at OWASP AppSec EU (AppSecEU) (2010).*





FACULTY OF ENGINEERING SCIENCE  
DEPARTMENT OF COMPUTER SCIENCE  
SCIENTIFIC COMPUTING GROUP  
Celestijnenlaan 200A box 2402  
B-3001 Heverlee

