ARENBERG DOCTORAL SCHOOL
FACULTY OF ENGINEERING

Towards a Secure Web:

Critical Vulnerabilities and Client-Side Countermeasures

Nikolaos Nikiforakis

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

August 2013

Towards a Secure Web:

Critical Vulnerabilities and Client-Side Countermeasures

Nikolaos NIKIFORAKIS

Supervisory Committee:
Prof. Dr. ir. A. Bultheel, chair
Prof. Dr. ir. W. Joosen, supervisor
Prof. Dr. ir. F. Piessens, co-supervisor
Prof. Dr. ir. B. Preneel
Prof. Dr. ir. D. Hughes
Dr. ir. L. Desmet
Prof. Dr. J. Caballero
(IMDEA-Software, Madrid, Spain)
Prof. Dr. G. Vigna
(University of California, Santa Barbara, USA)
Dr. ir. Y. Younan
(SourceFire, USA)

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

August 2013

© 2013 KU Leuven, Science, Engineering & Technology
Uitgegeven in eigen beheer, Nikolaos Nikiforakis, 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 andere wijze ook zonder voorafgaandelijke 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 978-90-8649-650-1
D/2013/10.705/65

Abstract

As the web keeps on expanding, so does the interest of attackers who seek to exploit users and services for profit. The last years, users have witnessed that it is hard for a month to pass without news of some major web-application break-in and the subsequent exfiltration of private or financial data. At the same time, attackers constantly register rogue domains, using them to perform phishing attacks, collect private user information, and exploit vulnerable browsers and plugins.

In this dissertation, we approach the increasingly serious problem of cybercrime from two different and complementary standpoints. First, we investigate large groups of web applications, seeking to discover systematic vulnerabilities across them. We analyze the workings of referrer-anonymizing services, file hosting services, remote JavaScript inclusions and web-based device fingerprinting, exploring their interactions with users and third-parties, as well as their consequences on a user's security and privacy. Through a series of automated and manual experiments we uncover many, previously unknown, issues that could readily be used to exploit vulnerable services and compromise user data.

Second, we study existing, well-known, web application attacks and propose client-side countermeasures, that can strengthen the security of a user's browsing environment without the collaboration, or even awareness, of the web application. We propose countermeasures to defend against session hijacking, SSL stripping, and malicious, plugin-originating, cross-domain requests. Our countermeasures involve near-zero interaction with the user after their installation, have a minimal performance overhead, and do not assume the existence of trusted third-parties.

Samenvatting

Het web blijft zich uitbreiden en dit geldt ook voor de interesse van cybercriminelen in het uitbuiten van gebruikers en diensten voor winstbejag. De laatste jaren zijn er nauwelijks maanden geweest zonder nieuws over een omvangrijke aanval op een webapplicatie en de daaropvolgende diefstal van private of financiële gegevens. Tegelijkertijd registreren aanvallers voortdurend bedrieglijke domeinnamen voor phishing, het verzamelen van private gebruikersgegevens en het uitbuiten van kwetsbare browsers of plugins.

In dit proefschrift benaderen we het steeds nijpender wordend probleem van computercriminaliteit vanuit twee verschillende en complementaire uitgangspunten. Ten eerste onderzoeken we grote groepen webapplicaties om er systematische kwetsbaarheden in te ontdekken. We analyseren de werking van referrer-anonymizing services, file hosting services, remote JavaScript inclusion en web-based device fingerprinting om hun interacties met gebruikers en derden te bestuderen alsook hun impact op de veiligheid en privacy van de gebruiker. Door middel van een reeks automatische en handmatige experimenten ontdekken we vele, tevoren onbekende, problemen die direct gebruikt kunnen worden voor het uitbuiten van kwetsbare diensten en het stelen van gebruikersgegevens.

Ten tweede bestuderen we welbekende bestaande aanvallen op webapplicaties en stellen client-side tegenmaatregelen voor die de beveiliging van de gebruikers surfomgeving versterken zonder de medewerking, of zelfs het medeweten, van de web applicatie. De voorgestelde tegenmaatregelen beschermen tegen session hijacking, SSL stripping en kwaadaardige cross-domain requests afkomstig van plugins. Deze tegenmaatregelen vergen na installatie vrijwel geen interactie met de gebruiker, hebben een minimale impact op de performantie en zijn niet afhankelijk van te vertrouwen derde partijen.

Acknowledgments

It is somewhat of a fact that, the “Acknowledgments” section of a dissertation is the part that’s read the most, simply because it is accessible to a much wider audience than the rest of the text. Thus, in this part, I would like to take the time to thank everyone who supported me, in one way or another, during the course of my PhD.

First and foremost, I am grateful to my supervisors, Wouter Joosen and Frank Piessens, for the support, guidance, and encouragement that they showed me over the past four years. It is a great privilege to be in an environment where you are given the opportunity to discover what you’re best at, and then the freedom to pursue it.

Second, I want to thank the people in my jury, Bart Preneel, Danny Hughes, Lieven Desmet, Juan Caballero, Giovanni Vigna and Yves Younan, for reading my dissertation and providing me with many helpful comments that greatly improved this final version. I also want to thank Adhemar Bultheel for chairing the jury.

Next, I would like to thank some of my friends and colleagues at the department. I am grateful to Theofrastos Mantadelis (also known as “Teo”) for his company and for introducing me to people and places when I first arrived in Leuven. Mainly because of him, I had the chance to meet Nima Taghipour (table tennis player extraordinaire), Wannes Meert, Gitte Vanwinckelen, Dimitar Shterionov, Koosha Paridel, as well as the “Greek crowd” outside of the department, Christos Varsakelis, Nina Siouta, Michalis Spyrantis, Alice Dillon and many more people, all of whom made me smile, and some of whom made a lasting impression in my life.

I am thankful to past and present office mates, Yves Younan, Kristof Verslype, Nataliia Bielova, Francesco Gadaleta, Jan Tobias Mühlberg, Milica Milutinovic, Job Noorman and Pieter Agten, for the all the fun, as well as serious, discussions that we have had in office number 03.127. Outside the confines of the third floor,

and at the risk of forgetting someone, I also want to thank Willem De Groef, Philippe De Ryck, Raoul Strackx, Ilya Sergey, Pieter Philippaerts, Dominique Devriese, Dimiter Milushev, Riccardo Scandariato and Steven Van Acker for our discussions over papers and warm cups of coffee. Steven is the person with whom I collaborated the most and I have a lot of respect for his skill-set. If a cure for cancer could be discovered by chaining Linux commands together, Steven would be the first one to find it. I want to thank everyone in DistriNet and the Computer Science Department who have contributed in making Celestijnenlaan 200A, such a peaceful and friendly working environment. I am also thankful for the support that I received from the EU-funded FP7 project NESSoS (NoE contract n. 256980), which helped me to focus on my research.

Apart from all the colleagues and friends in Belgium, I want to thank Giovanni Vigna, Christopher Kruegel and all the members of the SecLab in Santa Barbara for having me in the summer of 2012 and making me feel as part of the group. I also want to thank Martin Johns, Alexandros Kapravelos, Luca Invernizzi, Marco Balduzzi, Sebastian Lekies and John Wilander for being excellent remote collaborators. In addition to remote collaborators, I want to thank past colleagues and friends from Greece, namely Spyros Ligouras, Paris Amerikanos, Zissis Konstas, Giorgos Vasiliadis, Antonis Papadogiannakis, Vasilis Pappas, Sotiris Ioannidis and Evangelos Markatos, for their continuous support and encouragement.

I could not have made it this far without the love and support of my parents Theodoros and Aikaterini who provided for me, very often sacrificially. I thank Giannis, my brother, for loving me even though I was not always present to support him and help him out, and Dorothy, my aunt, for being more like a second mother to me. I am also grateful for the love and support of my late grandfathers Giannis and Nikos, my grandmother Mary, and my grandmother Thaleia. To all of them, I want to say:

Σας αγαπώ και σας ευχαριστώ για όλα.

When I came to Belgium in August of 2009, I did not expect that I would shortly thereafter meet the woman who is now my wife, Freya De Leeuw. Her love, support, and care, help me to move forward every single day. Even if my PhD would not have worked out, Freya was well worth my moving to Belgium. I am a blessed man to be able to call her my wife.

Anyone who knows me, also knows that my belief in God is integral to who I am. As such, lastly, I want to thank Him, the Father of lights, the Author of

creation, whose magnificent glory resonates through all the universe, for his unbounded mercy towards me.

Nick Nikiforakis
Leuven, August 2013

Contents

Contents ix

List of Figures xvii

List of Tables xix

1 Introduction 1

1.1 Cybercrime in Numbers 2

1.2 Willingness to Trust 2

1.3 Browser Security 4

1.4 Dissertation Scope 5

1.4.1 Discovering Critical Threats in Web Applications 6

1.4.2 Client-Side Mitigations for Web Application Vulnerabilities 6

1.5 Contributions 7

1.6 Outline of the Dissertation 8

1.6.1 Part I 8

1.6.2 Part II 9

I Discovering Critical Threats in Web applications 11

2 Referrer-Anonymizing Services 17

2.1 Introduction 17
2.2 Background 19
2.2.1 Referrer Header 19
2.2.2 Referrer Use-cases 20
2.2.3 Referrer Abuse-cases 21
2.3 Referrer-Anonymizing Services 22
2.4 Taxonomy of RASs 24
2.4.1 Redirection mechanism 25
2.4.2 Delay and Advertising 25
2.4.3 Mass Anonymization 26
2.4.4 Background Activity 27
2.5 Information Leakage 28
2.5.1 Experimental Setup 29
2.5.2 Results 29
2.6 User Categorization 30
2.6.1 Ethical considerations 31
2.6.2 Hiding advertising infrastructures 32
2.6.3 Remote image linking 33
2.6.4 Web-mashups 33
2.7 Tracking of Anonymizers 34
2.8 Related Work 36
2.9 Conclusion 38

3 File Hosting Services 39

3.1 Introduction 39

3.2 Life cycle of files on File Hosting Services 40
3.3 Privacy study 41
3.4 HoneyFiles 48
3.5 Countermeasures 51
3.6 Ethical Considerations 52
3.7 Related Work 53
3.8 Conclusion 54

4 Remote JavaScript Inclusions 55

4.1 Introduction 55
4.2 Data Collection 57
4.2.1 Discovering remote JavaScript inclusions 57
4.2.2 Crawling Results 59
4.3 Characterization of JavaScript Providers and Includers 61
4.3.1 Evolution of remote JavaScript Inclusions 61
4.3.2 Quality of Maintenance Metric 64
4.3.3 Risk of Including Third-Party Providers 67
4.4 Attacks 69
4.4.1 Cross-user and Cross-network Scripting 70
4.4.2 Stale Domain-name-based Inclusions 70
4.4.3 Stale IP-address-based Inclusions 72
4.4.4 Typosquatting Cross-site Scripting (TXSS) 72
4.5 Countermeasures 74
4.5.1 Sandboxing remote scripts 74
4.5.2 Using local copies 77
4.6 Related Work 78
4.7 Conclusion 80

5 Web-based Device Fingerprinting 81

5.1 Introduction 81

5.2 Commercial Fingerprinting 84

5.2.1 Fingerprinting through popular plugins 86

5.2.2 Vendor-specific fingerprinting 87

5.2.3 Detection of fonts 87

5.2.4 Detection of HTTP Proxies 89

5.2.5 System-fingerprinting plugins 90

5.2.6 Fingerprint Delivery Mechanism 92

5.2.7 Analysis Limitations 93

5.3 Adoption of fingerprinting 93

5.3.1 Adoption on the popular web 93

5.3.2 Adoption by other sites 94

5.4 Fingerprinting the behavior of special objects 96

5.4.1 Experimental Fingerprinting Setup 96

5.4.2 Results 97

5.4.3 Summary 104

5.5 Analysis of User-Agent-Spoofing Extensions 104

5.6 Discussion 108

5.6.1 Reducing the fingerprintable surface 108

5.6.2 Alternative uses of fingerprinting 109

5.7 Related Work 110

5.8 Conclusion 111

II Mitigations for known Web application vulnerabilities 113

6 Session Hijacking 117

6.1 Introduction 117
6.2 Background 119
6.2.1 Session Identifiers 119
6.2.2 Cross-Site Scripting attacks 119
6.2.3 HTTP-Only and Sessions 121
6.3 SessionShield Design 122
6.3.1 Core Functionality 123
6.3.2 Naming Conventions of Session Identifiers 123
6.3.3 Statistical Characteristics of session identifiers 124
6.4 Evaluation 125
6.4.1 False Positives and False Negatives 125
6.4.2 Performance Overhead 126
6.5 Related Work 127
6.6 Conclusion 129

7 SSL stripping attacks 131

7.1 Introduction 131
7.2 Anatomy of SSL stripping attacks 132
7.2.1 Redirect Suppression 133
7.2.2 Target form re-writing 133
7.3 Effectiveness of the attack 134
7.3.1 Applicability 134
7.3.2 Software feedback 135
7.4 Automatic Detection of SSL stripping 136

7.4.1 Core Functionality 136
7.4.2 Architecture of HProxy 136
7.4.3 Detection Ruleset 139
7.4.4 Redirect Suppression Revisited 142
7.5 Discussion 143
7.5.1 JavaScript Preprocessing 143
7.5.2 Signed JavaScript 144
7.5.3 Inspecting Client Requests vs. Server Responses 144
7.6 Evaluation 145
7.6.1 Security Evaluation 146
7.6.2 False Positives 146
7.6.3 Performance 148
7.7 Related work 149
7.8 Conclusion 150

8 Malicious, Plugin-Originating, Cross-domain Requests 151

8.1 Introduction 151
8.2 Technical background 153
8.2.1 The Same-Origin Policy 153
8.2.2 Client-side Cross-Domain Requests 153
8.2.3 An Opt-in Relaxation of the SOP 154
8.2.4 Client-side cross-domain requests with Flash 154
8.3 Security Implications of Client-Side Cross-Domain Requests . . 155
8.3.1 Vulnerable Scenario 1: Insecure Policy 156
8.3.2 Vulnerable Scenario 2: Insecure Flash Proxies 156
8.3.3 Resulting malicious capabilities 157
8.3.4 General Risk Assessment 158

8.4 Real-World Vulnerabilities 158

8.4.1 Deal-of-the-day Website: Insecure wildcard policy . . . 159

8.4.2 Popular sportswear manufacturer: Vulnerable Flash proxy 159

8.5 Client-Side Detection and Mitigation of Malicious Cross-Domain Requests 160

8.5.1 High-level Overview 160

8.5.2 Disarming potentially malicious Cross-Domain Requests . 161

8.5.3 Detailed Detection Algorithm 161

8.6 Evaluation 165

8.6.1 Security 165

8.6.2 Compatibility 166

8.6.3 Performance 167

8.7 Related Work 169

8.8 Conclusion 171

9 Conclusion 173

9.1 Summary 173

9.2 Recent Related Work 175

9.3 Towards Accurate and Systematic Experiments 177

9.3.1 Storage versus Reproducibility 177

9.3.2 User Identification 178

9.4 Future Work and Concluding Thoughts 178

A Experimental Parameters 181

A.1 SessionShield parameters 181

A.2 Quality of Maintenance metric 182

Bibliography 183

Curriculum Vitae **203**

List of publications **205**

List of Figures

2.1 HTTP messages involved in the use of a Referrer-Anonymizing Service 23

2.2 Weekly number of anonymization requests that our RAS received during our study 31

2.3 Facebook's main page when visited through a specific RAS . . 35

3.1 Length of the Identifier 45

3.2 Size of the Identifier's Character Set 45

4.1 Relative frequency distribution of the percentage of top Alexa sites and the number of unique remote hosts from which they request JavaScript code 59

4.2 Evolution of remote JavaScript inclusions for domains ranked in the top 10,000 from Alexa. 61

4.3 Cumulative distribution function of the maintenance metric, for different datasets 68

4.4 Risk of including third-party providers, included in high and low maintenance web applications. 69

5.1 The same string, rendered with different fonts, and its effects on the string's width and height, as reported by the Google Chrome browser 89

5.2 Fingerprinting libraries take advantage of Flash's ability to ignore browser-defined HTTP proxies to detect the real IP address of a user . . . 91

5.3 The top 10 categories of websites utilizing fingerprinting 95

5.4 A comparison between how many distinguishable feature sets and minor Google Chrome versions we have per Google Chrome's major versions. . . . 103

5.5 Feature-based fingerprinting to distinguish between Google Chrome major versions . . . 103

6.1 Average download time of the top 1,000 websites when accessed locally without a proxy, with a simple forwarding Python-proxy and with SessionShield . . . 126

7.1 Architecture of HProxy . . . 137

7.2 Example of an injected HTML form by a MITM attacker 141

7.3 Portion of the JavaScript code present in two consecutive page loads of the login page of Twitter. The underlined part is the part that changes with each page load . . . 145

7.4 False-positive ratio of HProxy using three different methods of whitelisting JavaScript . . . 147

7.5 Average load time of the top 500 websites of the Internet when accessed locally without a proxy, with a simple forwarding proxy(TinyHTTPProxy) and with HProxy . . . 148

8.1 General Use Case . . . 155

8.2 Vulnerable Flash Proxy . . . 155

8.3 Detection and Mitigation Algorithm . . . 162

List of Tables

2.1 Alexa Ranking of 30 tested RASs – gray color denotes RASs showing ads 24

2.2 Common features of Referrer-Anonymizing Services 26

3.1 Analysis of the Download URI's identifier 43

3.2 Number of files with sensitive types reported as private by our privacy-classification mechanism 44

3.3 Experiment on the non-sequential identifiers 46

3.4 Security features 46

3.5 Set of files containing fake data that were used as bait in our HoneyFiles experiment. The third column shows the resulting download ratio of each file by attackers 48

3.6 Attack geolocation recorded by our HoneyFile monitor 50

4.1 The ten most popular remotely-included files by the Alexa top 10,000 Internet web-sites 58

4.2 Evolution of the number of domains with same and new remote JavaScript inclusions for the Alexa top 10,000 62

4.3 Number of new domains that are introduced every year in remote inclusions. 62

4.4 Up-to-date versions of popular web servers, at the time of our experiment 66

4.5 Results from our experiment on expired remotely-included domains 72

4.6 Examples of mistyped domains found in remote JavaScript inclusion tags . . . 73

4.7 JavaScript functionality used by the 100 most popularly included remote JavaScript files . . . 75

5.1 Taxonomy of all features used by Panopticlick and the studied fingerprinting providers - shaded features are, in comparison to Panopticlick, either sufficiently extended, or acquired through a different method, or entirely new . . . 85

5.2 Differences in the order of `navigator` objects between versions of the same browser . . . 99

5.3 Unique methods and properties of the `navigator` and `screen` objects of the four major browser-families . . . 100

5.4 List of user-agent-spoofing browser extensions . . . 105

5.5 Standard properties of the navigator object and their values across different browser families . . . 106

6.1 Statistics on the usage of HTTP-Only on websites using session identifiers, sorted according to their generating Web framework 122

6.2 Default session naming for the most common Web frameworks 124

8.1 Nature of requests observed by DEMACRO for Alexa Top 1k websites . . . 167

8.2 Best and worst-case microbenchmarks (in seconds) of cross-domain requests . . . 167

A.1 Points used, as part of our QoM metric, for enabled, client-side, defense mechanisms and signs of good maintenance . . . 182

Chapter 1

Introduction

> "A computer lets you make more mistakes faster than any invention in human history – with the possible exceptions of handguns and tequila."
>
> MITCH RATLIFF

The near-ubiquitous presence of computers and Internet access has changed the way we communicate, work and think. People are expected to be computer literate, i.e., to be able to use computers efficiently, and more and more traditional businesses realize that through the use of information technologies, work can be done faster and cheaper. Due to the increased competition among registrars and web hosting companies, owning a domain name and renting the necessary hosting can currently cost as little as 100 euros per year, allowing many small, non-IT businesses, to not only use computers in their work, but to also create their own websites, advertising their goods and services. A survey by Netcraft in 2012, confirms this by reporting that there were over 640 million active websites [118].

Yet because of the exponential growth and constant innovation of IT and IT-related technologies, the knowledge and skill-set of people that work with computers and the Internet can vary dramatically. This becomes particularly important when one considers that the amount of crimes committed through the Internet is constantly rising.

1.1 Cybercrime in Numbers

Once upon a time, hackers were motivated by curiosity and by the “bragging rights” they would get if they would successfully find a way to use a product or service in a way, not intended by its makers. For example, the first widespread computer worm, the Morris worm, did little more than merely replicating itself on every successfully compromised machine, and using it as a launchpad for further exploitations [175].

Today, while there are still people who hack for the same reasons, the majority of attacks are politically or financially motivated. For instance, in 2010, Stuxnet was discovered, a computer worm which was created specifically to attack Iran’s nuclear facilities [49]. Stuxnet marked the beginning of highly-specific malware targeting industrial systems and the complexity of its operations, led researchers to believe that it was created with the backing of at least one government, for instance the one of the United States, rather than using the time and resources of individuals.

According to Symantec’s Norton cybercrime report for 2011 [130], cybercrime costed 388 billion USD of which 29.38% was a direct cash cost to people and businesses, i.e., money stolen from bank accounts, credit cards as well as spent while trying to resolve cybercrime. 73% of Americans reported to have experienced cybercrime at least once in their lifetime yet at the same time, 34% of those admit that they do not have up-to-date security software installed on their computers. Researchers also recently realized that cybercrime and malware infections are no longer a one-man operation. Today, there exist specialized underground markets where attackers rent out botnets for spamming and DDoS purposes, malware authors can purchase malware installations on victim machines, and exploit-kits can be bought, or rented by the day [58, 24].

1.2 Willingness to Trust

The aforementioned number comparison between the percentage of users who have experienced cybercrime versus the ones who have up-to-date security software installed, is an indicator of the inconsistency between the dimension of the cybercrime problem and the ways people deal with it. While there are some people who still go to great lengths to ensure their online security and privacy, for instance through the use of dedicated browsers, virtual machines, non-reusable passwords and anonymous routing networks, unfortunately, the majority of users are generally complacent. This complacency is sometimes the result of a lack of knowledge, but it is often coupled with claims about having

"nothing to hide" and pointing out that they are not important enough to be targeted. Both claims, however, are erroneous, with the former based on faulty definitions of privacy [172] and the latter based on misinformation, ignoring the fact that for every Stuxnet-like attack, there exist thousands that are conducted blindly, against all possible targets, in order to maximize the probability of discovering vulnerable targets.

Among the entire Internet population, one can readily observe that people belonging to the modern generation of Internet users tend to be very liberal about their data. Facebook, the largest online social network, has surpassed one billion user accounts and according to estimates of 2012, five new profiles are created every second [131]. Users constantly upload more and more pictures to Facebook, totalling 300 million new photo uploads per day. These numbers are staggering. They show that not only the majority of web users have made social networks part of their daily lives, but that they trust ever increasing amounts of personal data with third-party companies that seek to make profit through advertising. The same observation can be made about other free and popular online services, such as maps, email, and search. It is worthwhile to remember that there have been several cases, where hackers broke into celebrities' email accounts and ex-filtrated naked pictures and videos [137]. While this is of course illegal and punishable by law, it is important to pause and consider that these celebrities trusted large online companies employing hundreds of thousands of people, with their, arguably, most private data. Companies have also integrated Facebook into their products, having special interest groups, announcing news, and conducting sweepstakes, all over Facebook. In printed media, companies are advertising their Facebook web page right next to their own URL, tying themselves further to the image and popularity of one specific social network.

What is alarming about this trend is that it is not accompanied by a deeper understanding of how the Internet works and what are the best practices of staying secure on the web. This is further amplified by the fact that, in the last years, attackers are targeting end-users in addition to Internet-facing servers since the latter are becoming harder and harder to exploit. This is due to the advancements in low-level security and the deployment of multi-tier safety perimeters, such as firewalls and intrusion detection systems, on large corporate environments. At the same time, the user's Internet-reachable and potentially exploitable surface has expanded through the use of large, complex browsers, browsers plugins and other Internet-facing applications. These are, in turn, multiplied with every networked device (e.g. smartphones, tablets, laptops and desktops) that users may own.

There are several case studies that paint a troubling picture concerning online security, such as that 70% of people in a survey would trade their credentials for a candy bar [17], or many celebrities requesting photos and videos to be

"taken off the Internet". Next to these, there have been many systematic studies that assess a user's understanding of fundamental security concepts, such as the ability to separate between a benign and a phishing page [37], or the handling of SSL errors, as manifested in a browser [181]. Users without a technological background have been found, in general, to be prone to error and thus abuse. For instance, many users rely on improper security cues that can be easily spoofed by an adversary (such as the presence of a lock-icon somewhere in a browser's address bar), or, have formed a daily routine that is efficiency-based and does not account for security errors, e.g., the clicking of "OK" without a thorough reading of the relevant message, in order to proceed with the desired task [206]. On the privacy front, studies have found that attackers can join a victim's social network using very simple trickery, e.g., through the use of attractive profile pictures or by pretending to have common hobbies and interests.

1.3 Browser Security

Assuming a non-tech-savvy user, the only barrier left between a (possibly malicious) website and a user's private data are web browsers. Browsers are the all-encompassing tools of the modern web and many companies envision an Internet where everything is delivered to end users through their browsers, including the applications that users can utilize to create more data, e.g., web-based office suites, photo editors and email clients.

Today, browsers are significantly more secure than they were five years ago, due to the introduction of privilege separation, sandboxing and bug-bounty programs [147, 57]. Browsers vendors, however, still face the usual security versus usability dilemma. If a browser decided to be very strict, i.e., stricter than the rest of the popular browsers, in the interest of safe-guarding a user's security and privacy, it would most likely "break" many non-malicious websites, whose web programmers simply did not know any better. Many of the users of these browsers, who browse these insecure websites would witness the missing functionality, and instead of blaming the site for not being coded properly, they would blame the browser and adopt one of the less secure ones which would allow the insecure website to function properly. Thus, a browser vendor that decides to invest in stricter security measures, will most likely suffer a loss of their market share, at least in the short term. Given the fierce battle between browsers vendors for a larger market share, there is simply no way that a browser vendor would accept such a compromise. The result of this is that browser-security moves at the pace of the least-common denominator, where most browsers implement the same basic security measures (e.g., the

Same-Origin Policy [162]), and any advance measures are strictly opt-in (e.g., Content Security Policy [109]).

Most modern browsers support add-ons, in the form of plugins and extensions, that can significantly increase the browser's out-of-the-box functionality. Mozilla Firefox was the first popular browser that supported plugins, and today it still has the largest add-on market with almost 500 million plugins currently in use by Firefox users [51]. Among the wide range of extra functionality implemented by the offered add-ons, there are also many that enhance a user's security and privacy without relying on websites to opt-in. Apart from efforts from the industry, e.g., plugins by antivirus vendors, and volunteer extension programmers, academic researchers also took advantage of the extendibility of browsers and proposed a series of add-ons that solve, to some degree, one or more client-side vulnerabilities, such as Cross-Site Request Forgery [164, 35] and Cross-Site Scripting [81].

It is worthwhile to note that, at the time of this writing, three out of the top ten Firefox add-ons are security and privacy related. While this gives hope about a user's understanding of privacy, it is important to see it, in light of the total 450 million Firefox installations. For instance, NoScript, an extension with a clear security purpose is currently installed by approximately two million users, i.e., 0.44% of the total Firefox user-base. This shows that, even though extensions can definitely amplify a user's security, they are of no use to the majority of people who do not know about them and do not possess enough technical knowledge to discover them, install them and appropriately configure them. Thus, any client-side solutions that need user involvement must somehow be accompanied by an effort to educate the user about the problem that they tackle and the protection that they offer, making widespread adoption non-trivial. Moreover, history has shown that simple, understandable mechanisms that tackle an issue in a clear, albeit limited, way are preferred over complex systems with cross-cutting changes. This holds true even if the latter, when configured and applied properly, may be more secure than the former.

1.4 Dissertation Scope

When one takes into account the rapid growth of IT technologies, the growing number of people and companies who invest time and resources to take advantage of the Internet, and the ever increasing cybercrime, it becomes evident that research is necessary to identify current security and privacy issues, propose solutions for them, and to predict and tackle future problems before they are discovered and abused by attackers. Due to the sheer size of the web and

the plethora of different technologies, focal points are necessary to guide one's research. In this dissertation, we chose to focus on two complementary issues revolving around web security and privacy, particularly as these affect the user. The first part of our work is geared towards the systematic discovery of vulnerabilities that are prevalent across ranges of web applications. The second part proposes client-side countermeasures that defend against attacks which other researchers have identified, either through a first-hand discovery, or by identifying them when cybercriminals first used them.

1.4.1 Discovering Critical Threats in Web Applications

In Part I, we investigate clusters of web applications focusing on their architectural and implementation choices (both deliberate and accidental) that affect a user's security and privacy. Specifically, we investigate referrer-anonymizing services, file hosting services, the practice of remote JavaScript inclusions and the problem of user fingerprinting. These applications, their users, any third-parties involved, and the interactions between all the players, form ecosystems which have intricacies, specific to each type of web application. Through a series of automated and manual experiments involving code review, large-scale crawling, honeypot deployment, creation of our own web applications for gathering information about users, and analysis of all the resulting data, we show that in many cases the perceived security and privacy of such services is drastically different from the actual security and privacy. The purpose of this part is to identify the, previously unknown, security and privacy problems related to these services, so that future researchers may be able to accurately tackle these issues.

1.4.2 Client-Side Mitigations for Web Application Vulnerabilities

In Part II, we shift our attention to the client-side of web applications. As mentioned earlier, browser vendors hesitate to implement countermeasures that could break legitimate sites. Thus, users must rely on add-on functionality (delivered through extensions and plugins) to increase their security and privacy. Given the wide adoption of web applications, and their bolt-on nature, i.e., connecting various components on top of each other with the occasional unpredictable interactions, many security problems have been identified. Due to the importance of these problems, and the large potential for abuse, researchers have proposed many systems to protect users against them, with varying degrees of efficiency, completeness, and compatibility with existing websites.

The proposed systems, depending on their point of deployment, can be categorized into three categories: server-side, client-side, and hybrid. Server-side countermeasures are usually associated with the protection of the web server, even though there are many proposals for server-side systems that attempt to identify client-side attacks, ranging from modifications to server-side code generation to make code injection attacks harder [75], to reverse proxy systems that detect attacks against both the user [76] and the server [157]. Client-side systems, like the aforementioned browser plugins, are deployed only at the client-side and do not require the assistance of the webserver. Hybrid solutions, require the modification of both the server and the client in order to achieve their security and privacy goals. While the countermeasures belonging to the last category, due to their privileged position and access to both sides of the communication, can tackle the problems in the best way possible, they generally suffer from adoption and deployment issues since both browser vendors and web programmers must make the necessary changes and cooperate with one another.

In this part of our dissertation, we tackle some of the most recent and unaddressed web-application vulnerabilities identified by other researchers, by providing client-side security solutions in the form of security-enhancing proxies and browser extensions. To be precise, we propose, design, implement, and evaluate countermeasures against session hijacking, SSL stripping, and malicious, plugin-originating, cross-domain requests. We chose this category of countermeasures, due to their potential of adoption and their lack of server-side reliance. While we acknowledge that adoption by users is a non-trivial issue, we reason that having more options, i.e., more ways of staying secure, is better than having less.

1.5 Contributions

The main contributions of this thesis are situated in the two domains introduced above. Our chapters have been individually published in international conferences and workshops, on the following topics:

- Exploration and evaluation of the *modus operandi*, in the context of security and privacy, of:
  - Referrer-Anonymizing Services [128]
  - File Hosting Services [122]
  - Remote JavaScript inclusions [123]

  - Web-based device fingerprinting [125]

- Design of client-side countermeasures against:
  - SSL Stripping [129]
  - Session Hijacking [126]
  - Malicious, plugin-originating, cross-domain requests [92]

Apart from the work which was consolidated in the two aforementioned parts, we also researched other types of abuse and proposed countermeasures for non web-application-specific vulnerabilities. More precisely, we studied the adoption of bitsquatting [121], the security misconfigurations of web-hosting providers [124], proposed a countermeasure against buffer overflows happening in a process' heap [127], and created a tool for evaluating the protection of low-level countermeasures against attacks [209]. These results will not be discussed any further in this dissertation.

1.6 Outline of the Dissertation

1.6.1 Part I

In Chapter 2, we investigate referrer-anonymizing services and discover that, in many cases, the user's privacy is sacrificed for the privacy of websites that wish to hide their identity. Our work is the first one to systematically study these services and discover abuses.

In Chapter 3, we explore the privacy of file-hosting services and show that many of them have not only made poor implementation choices that allow attackers to gain access to private files belonging to users, but that attackers are already exploiting these weaknesses. We were the first to systematically study these services from a security and privacy point of view and to highlight the ongoing exploitation of sequential identifiers.

In Chapter 4, we investigate the practice of remote JavaScript inclusions, at a large scale, focusing on the security of remote code providers and the potential of attacking them as a way of reaching harder-to-get targets. Our work is the largest remote-JavaScript-inclusion study to date and the first one to discover four new types of vulnerabilities, related to remote JavaScript inclusions.

In Chapter 5, we shed light into the current practices of user fingerprinting by being the first to explore the workings of popular fingerprinting services, their adoption by modern websites and their effect on user privacy.

1.6.2 Part II

In Chapter 6, we describe session hijacking and propose a client-side countermeasure which detects session identifiers in a user's cookies and isolates them from the browser and any potentially malicious scripts. The novelty of our work, lies in the ability to reason about the confidentiality of cookies without the help of the web application.

In Chapter 7, we investigate SSL stripping and propose a client-side countermeasure which detects unauthorized page modifications by an active man-in-middle, in the user's local network. Our countermeasure is the first, client-side countermeasure against SSL stripping.

In Chapter 8, we propose the first, client-side countermeasure against malicious, plugin-originating, cross-domain requests which protects users by de-authenticating maliciously-generated requests.

Lastly, in Chapter 9, we summarize our work, propose areas for future work, and briefly discuss ways of setting up future, large scale experiments in order to achieve better reproducibility and accuracy.

Part I

Discovering Critical Threats in Web applications

Introduction and Problem Statement

"Furious activity is no substitute for understanding."

Rev. H.H. Williams

With every passing day, the Internet becomes more and more integrated in our society and our offline time continually decreases. Given the advances of data transfer protocols in cellular networks and the constant price decrease of smartphones and tablets, many people are almost always online. According to numbers of the first quarter of 2013, there are more than 2.2 billion Internet users who, in a single day, send almost 200 billion email messages and perform more than two billion web searches using Google's search engine [18]. These incredibly large numbers show that, for most people, Internet and the web are utilities that have become as common as electricity and water.

The evergrowing number of people joining the web, were first observed and then influenced by companies and service providers. The modern web of social networks, e-commerce, e-health, and e-banking, has little in common with the web of the 90s, i.e., a small population of static webpages that were, for the most part, the equivalent of web-accessible business cards. Today, companies are constantly migrating their traditional desktop applications to the web and entire operating systems have been constructed that "boycott" most native applications, favoring their web counterparts [212]. The increased connectivity and high bandwidth also allowed for an entirely new type of computing to emerge, i.e., cloud computing. Currently, thousands of companies use computing resources, e.g., computing time, network resources and storage, that are delivered as a service over the Internet.

The global accessibility of web applications, combined with the low entry cost for a new online business, in turn translated to hundreds or thousands of web

applications, all competing for the same users in the same market. For every popular house-hold-named web application, there exist scores of competing ones, trying to enlarge their user base by targeting users of specific backgrounds, ethnicities and interests. Eventually, the web applications of a certain popular category, together with the users that have clustered around them and the intended (and unintended) interactions between all the parties involved, create large online ecosystems.

Like in nature, the interactions of each ecosystem are complex and highly specific to that ecosystem. For instance, social networks that require a paid subscription are radically different than free ones, since the latter need to somehow generate sufficient income for their services. This, in turn, means that advertising companies become an integral part of that social networking ecosystem.

In the first part of this dissertation we explore and analyze four large ecosystems, focusing on the interactions between web applications, their users and third parties, that have security and privacy consequences. As we mentioned in Chapter 1, attackers are constantly on the lookout to abuse and exploit users and steal their data. Thus, vulnerabilities that affect a range of online services, rather than a single instance, are of particular interest to them. In this part, however, we do not limit ourselves to vulnerabilities that websites accidentally allow, but we also explore conscious security and privacy choices which may be beneficial for any single website, but have negative consequences for the security and privacy of their users. More specifically, we focus our attention to referrer-anonymizing services, file-hosting services, remote JavaScript inclusions, and the problem of user-fingerprinting.

In Chapter 2, we explore the workings of referrer-anonymizing services. These services are utilized by websites in order to increase the privacy of their users and hide their own identity, when transitioning from one website to another. While these services are very popular, i.e., they rank high on the Alexa list of top Internet websites [5], we were the first ones to systematically study them and explore the impact of their implementation choices on a user's privacy. Through a series of experiments, we demonstrate that, in many cases, the advertisers who are used to monetize these free services, abuse their power in order to discover the destinations of users and link them with their IP address, getting data about a user's browsing habits which they wouldn't be able to get, if a user did not make use of these "privacy-preserving" services. Moreover, we show that such services are also used by attackers when trying to hide the tracks of their phishing campaigns and are thus not well received by popular websites.

In Chapter 3, we study the privacy of 100 popular file-hosting services, i.e., services that assist users in exchanging large files over the Internet. Some of

these services had been studied before by Antoniades et al. [8], but not from a security and privacy perspective. We show that a significant percentage of these services implement their web applications in a way that allows attackers to discover and download files belonging to other users. Moreover, using a honeypot-like system, we demonstrate that attackers are indeed aware of these shortcomings and are actively exploiting them to gain access to user data. As one can plainly understand, the sensitive nature of files uploaded in such services means that a successful attack against them can have profoundly negative effects for their users.

In Chapter 4, we study the practice of remote JavaScript inclusions. The ease with which websites can embed JavaScript programs directly from third-party providers and the benefits of this "hot-linking" are compared to the possible security and privacy issues that arise when one blindly trusts remote hosts for JavaScript. Our study is the largest remote-inclusion study to date, with more than 3 million pages crawled and 8.5 million records of remote JavaScript inclusions discovered. The breadth of our study allowed us to reason about the issues of remote JavaScript inclusions with confidence as well as to discover four new instances of vulnerabilities, that were previously unknown.

Last, in Chapter 5, we explore the ecosystem of web-based device fingerprinting. Web-based device fingerprinting had received little attention, outside of the work of Eckersley [43] and Mayer [101], who showed that it is possible to uniquely identify users, through their browsing environments. We study three popular implementations of commercial fingerprinting services and highlight their workings, demonstrating how hundreds of thousands of users are unknowingly fingerprinted, on a daily basis by the sites they trust. We also quantify the protection of user-agent-spoofing browser extensions and show that, contrary to what previous research claimed [222], their use makes users more visible and more fingerprintable than before.

Given the size and ever-changing-character of the web's landscape, we can, by no means, claim to have explored all the important types of web applications. Our decision to focus on these four cases, was driven by their current relevance and their sustained popularity. At the same time, however, the specific methodology of systematically analyzing each ecosystem is reusable and we thus believe that future researchers can apply our methods when seeking to uncover privacy and security issues of new large-scale ecosystems, similar to the ones we have analyzed.

Chapter 2

Referrer-Anonymizing Services

Preamble

This chapter presents a systematic exploration of Referrer-Anonymizing Services with a focus on the mechanisms that affect a user's privacy. The contents of this chapter are replicated from the paper titled "Exploring the Ecosystem of Referrer-Anonymizing Services" [128], which was published in the proceedings of the 12th Privacy Enhancing Technology Symposium (PETS), 2012. This work was done with the collaboration of other authors from KU Leuven. Nick Nikiforakis was the lead author of this paper.

2.1 Introduction

In the infant stages of the Internet, privacy and anonymity were mostly unnecessary due to the small size of the online community and the public nature of the available data. Today however, this has changed. People have online identities, are connected to the Internet almost permanently and they increasingly store their sensitive documents, photos and other data online in the cloud. Our new online way of life provides interesting opportunities to those who seek to exploit it. In an extreme case, corrupt regimes trying to find out what their citizens are doing and thinking, want to violate both online privacy and anonymity [28]. The threat however, need not be a far-fetched scenario or

exclusive to the paranoid: large companies and organizations are also interested in the online habits of the masses for various reasons, e.g. targeted advertising.

Projects like The Onion Router (TOR) [38, 188] and the Invisible Internet Project (I2P) [66] provide online anonymity to their users by routing Internet traffic through a number of relays, thus making it harder for the endpoint to trace the source of the traffic. The application-layer however, on top of the network-layer where TOR or I2P reside, could still carry information that can compromise a user's anonymity or privacy. This is especially so when a web-browser is used, because browsers leak a wealth of information about their users. A study by the EFF's Panopticlick project [43] shows that, based on data typically provided by a browser, a web-site visitor can be uniquely identified in the majority of cases. Private details can be extracted even in the cases where users utilize their browsers' private modes [4] or spoof their user-agent information [50].

One particularly sensitive piece of data, transmitted with almost every HTTP request but commonly overlooked, is the referrer information in the 'Referer' [1] header, which can be used to trace the page where a visitor came from. Online services known as Referrer-Anonymizing Services (RASs) scrub this referrer information from HTTP requests, providing both anonymity to web-sites hosting links as well as privacy to users following those links. In this chapter, we take a closer look at RASs. We first perform a manual analysis of popular RASs and record their workings and architectural choices. Through a series of experiments we approach RASs from three different perspectives: the perspective of sites utilizing RASs, the RASs themselves, and the destination sites receiving traffic relayed through a RAS. In the first experiment, we determine what type of sites make use of a RAS and for what reason. The second experiment analyzes the data that RASs have access to and whether they actually protect the privacy of visitors and the anonymity of linking sites. In the last experiment, we observe the reactions of popular web-sites when they are exposed to incoming links relayed through a RAS. From these experiments, we can conclude that in several cases, user privacy is sacrificed for the linking site's anonymity and that not all RASs can be trusted with private data.

The main contributions of this chapter are:

- Large-scale study of RASs and their common features.
- Experimental evidence of privacy and anonymity violations from RASs.
- Identification of types of RAS users and the rationale behind their usage.

[1] The correct spelling is 'referrer'. The misspelled word 'referer' was introduced by mistake by Phillip Hallam-Baker [67] and later adopted into the HTTP specification.

- Analysis of third-party site responses towards traffic relayed through RASs showing that RAS-relayed traffic is occasionally not well-received.

2.2 Background

In this section we briefly go over the workings of the referrer header and we list some valid use-cases as well as abuse-cases for this header.

2.2.1 Referrer Header

In the HTTP protocol, all client-side requests and server-side responses have headers and optionally a data body. At the client-side, each request contains headers that, at minimum, ask for a specific resource from the web-server, in a `GET` or `POST` manner and in the context of a specific web-site (`Host`), since typically a single web-server serves more than just one web-site. On top of these headers, browsers add a wide range of other headers, the most common of which are headers specifying the user's cookies towards a specific web-site, the user-agent and the encoding-schemes accepted by the current browser.

An HTTP header that is less known but as present in requests as all aforementioned headers, is the '`Referer`'. The HTTP referrer header is automatically added by the browser to outgoing requests, and identifies the URI of the resource from which the current request originated [151]. For instance, if a user while being on `www.example.com/index.php?id=42`, clicks on a link to `www.shopping.com`, her browser would emit a request similar to the following one:

```
GET / HTTP/1.1
Host: www.shopping.com
User-Agent: Mozilla/5.0 (X11; Linux i686)
Accept: text/html,application/xhtml+xml
Proxy-Connection: keep-alive
Referer: http://www.example.com/index.php?p=42
```

In this request, the user's browser provides to `www.shopping.com` the exact location of the page containing the clicked link, resulting in the request towards their servers. This behavior is true not only when a user clicks on a link, but also on all the non-voluntary requests that a browser automatically initiates while parsing a page. For example, all requests created while fetching remote images, scripts, cascading style sheets and embedded objects will contain the referrer header. The referrer header is traditionally omitted in one of the following

cases: *(i)* when users manually type a URI in their browser's address bar, *(ii)* when users click on an existing browser bookmark and *(iii)* when users are on a HTTPS site and click on an HTTP link.

In HTML5, the web-programmer can add a special 'noreferrer' attribute to selected anchor link tags that will cause the browser not to emit the referrer header when these links are clicked [207]. At the time of this writing, from the most popular three browsers (Mozilla Firefox, Google Chrome and Internet Explorer), Google Chrome is the only browser which supports this new 'noreferrer' attribute. We believe that this lack of browser support will only amplify the hesitation of web-developers in trying and adopting new security/privacy mechanisms [228]. For this reason, we do not expect widespread use of this 'noreferrer' attribute any time in the near future.

2.2.2 Referrer Use-cases

In this section we provide a non-exhaustive list of legitimate uses of the HTTP referrer for the web-server which receives the referrer-containing request:

- **Advertising programs**: In many cases, a web-site will buy banner/link placement space on more than one third-party web-sites. Using the referrer header, the advertised site can assess the percentage of visitors coming from each third-party site and use this information to either renew or cancel its advertising contracts.
- **CSRF Protection**: Cross-site Request Forgery (CSRF) is a type of attack where the attacker abuses the established trust between a web-site and a browser [226]. In the typical scenario, a victim who has an active session cookie with a web-application is lured into visiting a malicious site which initiates arbitrary requests towards that web-application in the background. The victim's browser appends the session cookie to each request thus validating them towards the web-server. Due to the way this attack is conducted, the referrer header in the malicious requests will not be the same as when the requests are conducted by the user, from within the web-application. Thus, a simple countermeasure against CSRF attacks is to allow the requests containing the expected referrer header and deny the rest.
- **Deep-linking detection**: Deep-linking or 'hotlinking' is the practice of linking directly to an object on a remote site without linking to any other part of the remote site's content. In practice, this behavior is unwanted, when the object that is deep-linked was originally given to users after a

series of necessary steps (e.g. giving access to a music file after filling out an online survey) [27]. By checking the referrer header before releasing the object, the site can protect itself from users who did not go through the expected series of steps. Unfortunately this approach can be easily circumvented by users who change the referrer header values of their requests to match the expected value of the remote site, using a modified browser.

- **Access-Control**: Using the same reasoning as in deep-linking, a site can enforce access control to individual pages by making sure that the visiting user arrives there only from other selected destinations. This technique is also used to provide special offers when a site is visited from another site that would normally not be visible to regular users. Wondracek et al. [211] discovered that this technique is used by traffic brokers in adult-content web-sites. As in the previous use case, this sort of access-control can be bypassed by a user with malicious intent.
- **Statistics gathering**: In large and complex web-sites, the web-developers seek to understand whether the content layout is facilitating the user into finding the data that they need. Through the use of the referrer, web-applications can track users between pages (without the need for cookies) and find the most common visitor paths.

2.2.3 Referrer Abuse-cases

The same referrer information that can be used for legitimate reasons, can be abused by web-site operators to assault a user's privacy and in certain cases even perform user impersonation attacks.

- **Tracking visitors**: Traditionally, users associate tracking with tracking cookies. A web-site that wishes to track its users between page loads, or collaborating sites that wish to track users as they transition from one to the other can do so through the referrer header even if users delete their cookies on a regular basis.
- **Session Hijacking**: As described in Section 2.2.1, the referrer header contains not only the domain of the page where each request originated but the full URI of that page. That becomes problematic when web-sites use GET parameters to store sensitive data, as is the case in sites that add session information to URIs instead of cookies. In this case, the HTTP referrer will contain the session identifier of the user on the originating site, allowing a malicious operator of the target web-site to impersonate the user on the originating site [113].

- **Sensitive-page discovery**: It is common for web-developers to hide the existence of sensitive files and scripts in directories that are accessible from the Web but are not linked to by any visible page of the web-site. These files or directories sometimes have obfuscated names to stop attackers from guessing them. Penetration-testing tools such as *DirBuster*[2] that attempt to guess valid directories using dictionary and brute-force attacks, attest towards this practice. In such cases, if one of the sensitive pages contains an external link or external resource, the exact location of that page can be leaked to the remote web-server through the referrer header.

2.3 Referrer-Anonymizing Services

In Section 2.2 we described the possible uses and abuses of the HTTP referrer. Given the wealth of information that the referrer header provides, one can think of many scenarios where the user doesn't want to release this header to remote web-servers.

Today, users can achieve this either by configuring their browser not to send the referrer header, or through the use of Referrer-Anonymizing Services when clicking on links from sensitive web-pages. While the former approach is available on many modern browsers, it works as an all-or-nothing setting in the sense that the user cannot selectively allow the transmission of the referrer header. This can be problematic when a user navigates to web-applications that use the referrer header as a CSRF countermeasure. In such cases, the user wouldn't be able to use the web-application, unless they re-enable the transmission of the referrer header. An additional problem is for web-site owners who wish to link to third-party sites but not at the expense of uncovering their identity. A controversial but popular example are 'warez forums'[3], where the descriptions of the pirated software or multimedia are usually given by linking back to the legitimate web-sites. In these cases, the web-site operators cannot rely on privacy-aware users, but must use a solution that will seamlessly work for all. This can be achieved through the use of Referrer-Anonymizing Services.

A Referrer-Anonymizing Service (RAS) is a web-service that is responsible for anonymizing the referrer header of a user before that user reaches a remote web-server. Note that, for security reasons, a web-site is not allowed to arbitrarily change a user's referrer header. The referrer header is created and emitted by the browser and thus the only way to anonymize this header is for the RAS to place itself between the site that links to an external resource, and

[2] `http://sourceforge.net/projects/dirbuster/`
[3] 'warez' is slang for pirated software

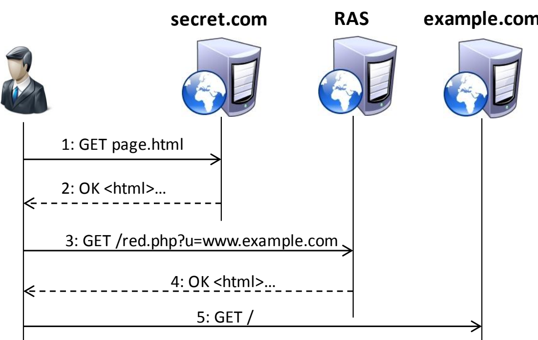

Figure 2.1: HTTP messages involved in the use of a Referrer-Anonymizing Service

that resource. By doing so, the RAS appears in the referrer header of the user's browser, instead of the original web-site, thus effectively anonymizing the original web-site. This technique is conceptually similar to the anonymizing techniques applied by Crowds [149] and TOR [38] where instead of the user's IP address, the link-embedding web-site is hidden.

Figure 2.1 shows the series of steps involved when using a RAS. In steps 1 and 2, a user requests and receives a page from `secret.com`. This page has a link that, if clicked, will eventually lead to `example.com`. However, since `secret.com` wishes to remain anonymous, it uses a RAS instead of linking directly to `example.com`. In step 3, the user clicked on the link expecting that it leads to `example.com`. However, the link creates a `GET` request towards a RAS with `example.com` as its argument. In response, the RAS generates a page (step 4) that will automatically redirect the user to `example.com` either directly, or after a timeout. In both cases, as far as the user's browser is concerned, the final request towards `example.com` originated not from `secret.com` but from the web-site of the RAS. Thus, the request depicted in Step 5, will have the redirect-causing web-page of RAS as its referrer, effectively hiding the original source of the link. Note that in the aforementioned process, `secret.com` will still reveal its presence to an external entity, but it chooses to reveal itself to the RAS instead of `example.com`.

The RAS can redirect the user's browser to `example.com` using one of the following ways:

- **HTTP MOVE messages**: When a web-server receives a request for a resource, it can emit a 301/302 `HTTP MOVE` message, that informs the user's browser of the 'move' of the resource, and provides it with the new

location. Upon the receipt of such a message, a browser automatically initiates a new request towards the instructed location, thus completing the redirect.

- **HTML Meta-Refresh tag**: One tag of the HTML specification allows the web-developer to ‘refresh’ a page after a configurable number of seconds. The refresh can load the same page, or a new one. For example, `<meta http-equiv="refresh" content="5;url=http://www.example.com/index.html">` instructs the user’s browser to replace the current page with the main page of `example.com` upon the expiration of 5 seconds.

- **JavaScript**: The same effect can be achieved using JavaScript, by setting the value of the `window.location` property to the desired site.

2.4 Taxonomy of RASs

In this section we analyze and compare common features of real-world Referrer-Anonymizing Services. We obtained a list of 30 functional RASs, shown in Table 2.1, by using a well-known search engine and searching for phrases related to their services, such as ‘referrer anonymization’ and ‘hiding referrer’. The popularity of some of these services is evidenced by their high ranking in the Alexa top sites list. For instance, the most popular RAS, `anonym.to`, currently ranks higher than well-known sites such as `blackberry.com` and `barnesandnoble.com`. We summarize the discovered features of the studied RASs in Table 2.2.

URI	Ranking	URI	Ranking	URI	Ranking
anonym.to	661	nolink.in	152,160	anonym2.com	846,517
hidemyass.com	904	cloakedlink.com	187,162	sock.im	933,195
referer.us	7,662	th3-0utl4ws.com	257,867	hidelinks.net	1,170,906
anonymizeit.com	10,212	savanttools.com	305,880	anon.projectarchive.net	2,591,907
lolinez.com	35,526	a.myurl.in	311,033	1-url.net	4,032,638
nullrefer.com	37,227	privatelink.de	338,512	crypo.net	5,009,009
linkblur.com	62,993	carcabot.com	347,260	anonym.ehmig.net	5,510,217
refblock.com	63,834	linkanonymizer.net	433,913	hidehelp.com	9,470,830
dereferer.ws	104,803	spoofurl.com	645,526	devgizmo.com	No Data
anonymous-link.net	118,261	refhide.com	679,101	theybetrollin.com	No Data

Table 2.1: Alexa Ranking of 30 tested RASs – gray color denotes RASs showing ads

2.4.1 Redirection mechanism

By manually visiting and recording the redirection mechanisms of the 30 RASs, we found out that 73% of them were redirecting their users using the meta-refresh mechanism, 20% using JavaScript and 7% using a combination of both 302 and meta-tags. The use of the HTML meta-refresh is the most common mechanism because it doesn't require JavaScript to execute and because it allows the RAS to delay the redirect in order to show advertising banners to the visiting users. The sites that used JavaScript to redirect a visitor, used it together with a timeout function to emulate the effect of the meta-refresh mechanism.

The HTTP MOVE messages were the least used among the services for two reasons. Firstly, redirects occurring through a 301/302 HTTP message retain the original referrer header and are thus not suited for use from RASs. The services that did utilize them, always combined them with a meta-header, where the 302 message would redirect the user to another page on the RAS's web-site which would then use an HTML meta-refresh tag. Secondly, even if the browser would clear out the referrer header, the HTTP MOVE mechanism doesn't allow for a delayed redirect, thus the services cannot use it to show advertisements.

An interesting observation is the diverse redirection behavior that different browsers display. When using Mozilla Firefox (version 9) and Google Chrome (version 16), a redirect implemented through JavaScript retains the referrer that caused the redirect. That is not a problem for RASs since the page that causes the redirect is not the original page that wishes to remain hidden, but a page of the RAS (step 4 in Figure 2.1). On the other hand, the same redirection mechanism in Microsoft's Internet Explorer 8, clears out the referrer. Contrastingly, Firefox and Internet Explorer clear out the referrer header in case of an HTML meta-refresh but Chrome still retains the redirect-causing referrer. From a point of user privacy, the complete clearing of the referrer is the best option for the user since the web-server cannot distinguish between users coming from web-sites that protect themselves and users who typed in the URIs or clicked on their browser bookmarks. However, the same mechanism that protects a user's privacy may negatively affect a user's security, as later explained in Section 2.8.

2.4.2 Delay and Advertising

Since all RASs that we encountered were providing their redirection services for free, there is a high probability that they attempt to capitalize on the number of incoming users through the use of advertising. From our set of 30

Common Feature	Percentage of RASs
Redirection:	
HTML meta-refresh	73%
JavaScript	20%
HTTP MOVE+ meta-refresh	7%
Ads	36.66%
Mass Anonymization	50%

Table 2.2: Common features of Referrer-Anonymizing Services

services, 11 (36.66%) were displaying advertising banners to the users waiting to be redirected to the destination web-site. From these services, 10 of them were constructing advertisements on the fly (through the use of client-side scripting and a chain of redirections) and only one had the same banners, statically embedded in its web-site. We also noticed that the sites that included advertising had, on average, a higher delay than the non-advertising web-sites which sometimes didn't delay the user at all. More specifically, the RASs with advertisements were redirecting the user after an average delay of 11.8 seconds whereas the non-advertising RASs were redirecting the user after an average delay of 2 seconds.

An interesting observation for the RASs that create dynamic advertisements is that all the requests towards the advertising agencies contain a referrer header which is the URI of the RAS page where the user is waiting to be redirected to the destination site. Since all RASs work by receiving the destination URI (`example.com` in Figure 2.1) as a GET parameter, the various advertising agencies get access, not only to the IP addresses of the RAS visitors, but also to their eventual destination. By combining this knowledge with other data, they may be able to associate users with sites, even if the destination web-site doesn't collaborate with a specific advertising agency. Thus, in one third of the cases, the privacy of individual users is sacrificed for the anonymity of the linking site.

2.4.3 Mass Anonymization

Most RASs have a simple API for use of their services. All a RAS needs, is a remote URI to which it will redirect users while clearing, or substituting, their referrer header. For this reason, all RASs work in a stateless fashion. Unlike URL shortening services, where a user needs to first visit the service and generate a new short URL for their long URL, a user can utilize a RAS without first visiting the RAS's web-site. In our example in Figure 2.1, the administrator

of `secret.com` can create an anonymized-referrer link to `example.com` simply by making a GET request with `example.com` in the `u` parameter.

This stateless nature of RASs allows for mass-anonymization of links without the hassle of registering each and every link to a remote service. From the 30 RASs that we analyzed, 50% were providing a mass-anonymization option through the use of an anonymizing script. This script, which is supposed to be included by remote web-sites, iterates over all `<a>` elements of the current HTML page and converts all links to RAS-links. Additionally, the scripts usually provide a white-list option where domains that do not need be anonymized (such as the links to local pages within a web-site) can be listed and excluded from the anonymization process. While we didn't encounter a misuse, a site including a remote anonymizing script is implicitly trusting the RAS providing it to not include malicious JavaScript along with the anonymizing functionality.

2.4.4 Background Activity

Apart from advertisements, RASs can use the browsers and IP addresses of visiting users to conduct arbitrary requests before redirecting the users towards their final destination. This activity can range all the way from harmless but unwanted to malicious. In our analysis of 30 RASs, we found that 2 services were performing unexpected actions that were hidden from the user.

The first service had embedded an invisible iframe that performed a search request with the keyword 'myautopass' using Google Search. While the RAS cannot steal any private data from that iframe (since the Same-Origin Policy disallows such accesses), the requests were made by the user's browser and user's IP address. As far as Google Search is concerned, tens of thousands of people [4] search for that word on a daily basis, an action which most likely affects the ranking and trend of that keyword, even if the shown links are never clicked.

The second service, instead of redirecting the user to the desired web-site, created a frameset with two frames. In the first frame, which spanned the entire page, it loaded the requested site and on the second frame it loaded a local page of that RAS. In this scenario, while the user gets access to the remote page, they never actually leave the site of the RAS. This 'sticky' behavior is common in anonymizing web-proxies which request a page from a remote server on the user's behalf and then present the result to the user. To the remote web-server, the request appears as coming from the anonymizing proxy's IP address, thus hiding the user's IP address. Note however that in the case of

[4] We obtained the RAS's estimated number of daily visitors using `quantcast.com`

RASs, this ‘sticky’ behavior adds no more privacy to the visiting user, since the requests are all made from the client-side and thus using the user’s browser and IP address.

By analyzing the contents of the second frame, we observed that through a series of redirects, the site was opening the registration page of a file-hosting web-site, with a specific affiliate identifier. It is unclear how the operators of that RAS were expecting to fill in the registration page in an invisible frame but this method hints towards the RAS’s attempt to capitalize on visiting users for monetary gains in affiliate programs.

2.5 Information Leakage

Whenever a user utilizes an online service, there is always the possibility that the service will retain information from the user’s activity and use that information in the future, possibly for financial gains. In Section 2.4.2 we showed that 36.66% of all tested Referrer-Anonymizing Services used ads as a way of getting monetary compensation for their free services. In almost all cases, the ads were created dynamically, initiating a GET request for a script or an image from the RAS to the advertising company. Through the referrer header, these requests reveal to the advertising agencies which page the user is on (the RAS waiting page) and the page that the user intends to go to (the destination GET argument given to the RAS). This is problematic for two reasons: first, if the destination URI contains a secret parameter tied to a specific resource (e.g. a session identifier, a file identifier for file-hosting services or a document identifier for online collaboration platforms) this identifier will be disclosed to an untrusted third party (the advertising agency). The second reason is that advertising agencies gain more information about users and the sites they visit even if the destination sites do not collaborate directly with them.

Another instance of the same problem is encountered between the source site and the RAS. The administrator of a RAS is able to view the referrer headers of traffic towards their service and can thus discover the original pages that relay visitors through them (e.g `secret.com` in Figure 2.1). If the source site hosted the link to the RAS on a page with sensitive data in its URL (both path and GET parameters) – e.g. `secret.com/admin.php?pass=s3cr3t` – this will be available for inspection to the utilized RAS.

In order to measure whether the various advertising agencies of RASs make use of users’ referrer headers and whether the administrators of RASs access sensitive source pages, we conducted the following experiments.

2.5.1 Experimental Setup

We first created and registered `fileleaks.co.cc`, a web-site supposedly providing leaked sensitive documents and then developed two crawlers that visited all the RASs daily and requested a redirection towards URIs within our site. e.g. `http://anonym.to?http://fileleaks.co.cc/index.php?filename=[POPULAR_TOPIC]&dlid=[PER_RAS_UNIQUE_ID]`. The `dlid` contained a random identifier that was unique for all tested RASs and allowed us to accurately detect which RASs leaked our destination site.

The first crawler was simply requesting the URI in a `wget`-like way and proceeding to the next RAS. In this case, our destination URI could be leaked only through the web-server logs of the target RAS since no scripts or images were rendered. The second crawler was actually an instrumented instance of Firefox that automatically visited each site and waited for 10 seconds before moving on to the next target. The key difference between the two crawlers is that the latter one was a functional browser which executed all needed image and JavaScript requests to fully render each page. These two crawlers allowed us to roughly distinguish between URIs leaked by the web-administrator of a RAS and URIs leaked through the referrer header sent to advertising agencies.

To detect RAS administrators looking for sensitive pages in the referrer headers of their logs, we added a fake password-protected administrative panel to our site and programmed an additional `wget`-like crawler which constantly visited all RASs, pretending that the request for anonymization was originating at `http://fileleaks.co.cc/admin/index.php?password=[SECRET_PASS]&pid=[PER_RAS_UNIQUE_ID]`. The fake administrative script was logging all accesses and the `pid` GET parameter was used to distinguish leakage between the tested RASs as in our first set of crawlers.

2.5.2 Results

Leakage of destination site:

In a 30-day period our monitors on `fileleaks.co.cc` recorded a total of 250 file requests using unique URIs that were made available to the Referrer-Anonymization Services as destination sites. By decoding each URI we identified that the URIs were leaked by 3 different RASs. Interestingly, all the recorded URIs were communicated to the services through our instrumented Firefox crawler and not through the `wget`-like crawler, implying that the URIs were most likely leaked by subsequent JavaScript and image requests of each RAS-

waiting page. For privacy and ethical reasons, we do not identify the services by name and we refer to them as RAS1, RAS2 and RAS3.

The unique URI of RAS1 was found in 1.2% of the requests. It appears that the service leaked the URIs directly to a specific search engine, which at a later time requested the files with the exact parameters originally provided to RAS1. RAS2 and RAS3 were both leaking the requests towards services running on Amazon's Elastic Compute Cloud. The requests leaked by RAS2 (88.8% of the total requests) were revealing, through their user-agent, that they were crawlers working on behalf of a specific advertising company which specializes in identifying which pages would prove to be the best ones for ad placement for any given product or service. The last set of requests (10% of the total) with leaked identifiers from RAS3 were also originating from hosts on Amazon's Cloud but their user-agent didn't provide any identifying details.

In total, our experiment showed that 10% of the tested RASs were, knowingly or not, leaking the referrer-header of their users to third-party advertising agencies who were recording them and using them at a later time. Given the risks associated with the leakage of a visiting user's IP address and destination site, we believe this to be a significant privacy risk.

Leakage of originating site:

In the same period, our administrative-panel monitor recorded three visits from the same visitor. In the third visit, the user provided the exact `password` and `pid` combination that our third crawler was providing one of the 30 tested RASs through its referrer header. It is important to realize that given the nature of our crawler, the referrer header containing the `fileleaks.co.cc` administrative panel URI (and password as GET parameter) could only be retrieved from the RAS web-server's logs since the pages were always retrieved but never rendered in a real browser. Thus, no advertising agencies, or legitimate Web traffic scripts could ever access our referrer header. This shows that the administrator of one of the thirty RASs was actively searching the logs of the RAS in an effort to identify 'interesting' source sites and then manually inspect them.

2.6 User Categorization

In the previous section, we explored the various features of RASs and recorded which traits are prevalent and for what reasons. While this gives us an insight of the motives behind these services, it doesn't provide any specific details on the users who use these services or the actual reasons justifying their usage.

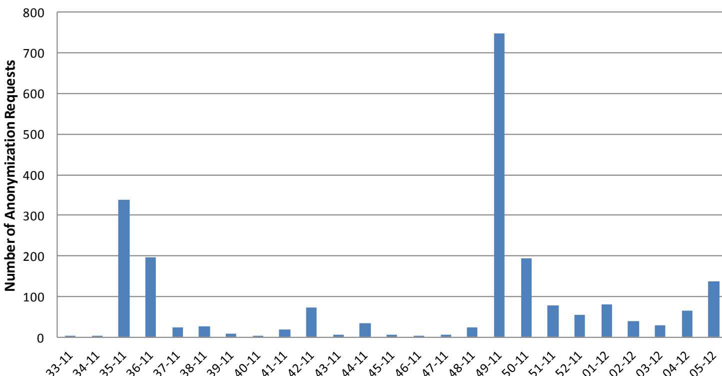

Figure 2.2: Weekly number of anonymization requests that our RAS received during our study

In order to investigate the user-part of the RAS ecosystem, we created and advertised our own Referrer-Anonymizing Service, which we made available at `www.cleanref.us`. In a period of 25 weeks, our service received a total of 2,223 requests for referrer-anonymization. Figure 2.2 shows that the weekly usage of our service varied significantly. In the next sections we describe a subset of these requests according to their purpose.

2.6.1 Ethical considerations

The data that was collected for this experiment are the following: For each request we recorded i) its timestamp, ii) the IP address of the host performing the request, iii) its GET parameters, and iv) the referring host. These were collected in a single text file on the web server, in a password-protected directory that only we had access to.

The data collected is a subset of the data collected by every web server on the web in standard server logs. Many web developers even share this collected information with third parties, such as Google Analytics, for the purpose of gathering usage statistics. The reason for deploying our own RAS was to identify potential abuse of such services. Since reporting to users up front that they would be part of an experiment would defeat the purpose of the experiment, our RAS did not explicitly warn users of the data collection. Because of the nature of the data collected, and the fact that these data are collected by every web server, we believe this lack of user warning to be acceptable.

2.6.2 Hiding advertising infrastructures

The greatest part of the first peek of Figure 2.2 corresponds to 317 requests conducted by 127 unique IP addresses. By analyzing our data, we discovered that for two days in the 35th week of 2011, our service was part of two advertising campaigns and the requests were from users who were the targets of these campaigns. While a significant number of countries was involved, the campaigns seem to have been targeting users mostly in Vietnam and the US, since these countries received 31.49% and 16.53% of the total traffic, respectively. It is unclear whether the advertising campaign was conducted through spam emails or through browser pop-up windows, however the fact that the advertiser used a Referrer-Anonymizing Service shows that they wished to conceal the exact method of driving traffic by hiding the referrer header from the final web-servers.

In our referrer logs for those days, we found 3 types of links. One type of link, was used to drive traffic directly from our service towards the destination web-site. The second type of link, was a chain of RASs all connected together in a way that allowed each RAS to redirect the user to next one, until the user is redirected to their final destination. For example, when the link:

```
http://cleanref.us/?u=http://www.refblock.com?http://cloakedlink.com
/zqkzqrfvgs
```

is clicked (or opened in a pop-up) by a user, her browser will request a page from `cleanref.us` which will redirect her to `refblock.us` which in turn will redirect the user to `cloackedlink.com`. `cloackedlink.com` is the end of the chain and when resolved, it will redirect the user to the actual site. The combination of multiple RASs allows the advertiser to hide its presence behind multiple layers of indirection. For instance, using this method, an inspection of the referrer does not reveal whether `cleanref.us` was the first part of a chain or whether another RAS redirected the user to our service using a redirection method that completely cleaned the referrer header. The last type of link, pointed to a service that received an obfuscated string as its only parameter:

```
http://linkfoobar.net/track-link.php?id=aHR0cDovL3d3dy5jbGVhbnJlZi51c
y8/dT1odHRw0i8vd3d3LnJlZmJsb2NrLmNvbT9odHRw0i8vY2xvYWtlZGxpbmsuY29tL3
pxa3pxcmZ2Z3M=
```

The `id` argument passed to the anonymized `linkfoobar.net` web-site is a Base64-encoded string that, when decoded, makes a chain of RASs similar to our previous example. As the name suggests, this is most likely the first part of

the chain where the advertiser first tracks that someone clicked on a tracked link and then proceeds to redirect the user to the final destination through a chain of RASs.

By combining the knowledge of all three categories of referrer URIs found in our logs, it is evident that the advertiser mixes the order of RASs in their chains in order to reveal only part of their traffic and infrastructure to each RAS. Thus in some cases, our service was the last in the chain of RASs, sometimes in the middle and occasionally the first service that began the chain of RASs after the user's click was registered by `linkfoobar.net`.

2.6.3 Remote image linking

Throughout our experiment we noticed several groups of requests (e.g. weeks 42-11, 04-12 and 05-12 in Figure 2.2) towards image files, some of them located on popular image hosting sites. By sampling some of the destination URLs, we noticed that the requested images were almost always of an adult nature. For the sampled requests, we also reviewed the referrer header when it was available. The observed linking sites fell into two categories. In the first category, the sites linking to adult images through our RAS were forums where the users could write new posts and include links to images. In this case, the RAS was added to hide the linking site from the image hosting site, since the uploaded images didn't conform with the rules of the latter. Some of the requests were using more than one RAS chained together as shown in Section 2.6.2.

In the second case, the linking sites were personal sites that were requesting images either from image hosting sites or directly from adult-content sites forming a client-side image collage. As in the former category, the owners of such pages were hiding the exact location of their personal pages from the sites hosting the linked images.

2.6.4 Web-mashups

A web-mashup is a combination of data and functionality from more than one remote service, which has more value than any one of the remote services by itself. The highest peek in Figure 2.2 stems from the adoption of our service from a book price-comparison web-application. The destinations of these requests are popular online bookstores and other price-comparison sites in Europe and the US. Each request contained the ISBN number of a different book. In a period of 9 weeks, the web-application initiated a total of 1,273 requests for anonymization of which 1,198 (94.10%) formed a unique combination of ISBN

number and third-party service. Given the vast majority of unique queries for books, we believe that the requests happen once and their results are cached in the web-application. The usage of a RAS in between the book price-comparison web-application and the other online bookstores, allows the former to retrieve information from the latter without revealing its location or purpose.

2.7 Tracking of Anonymizers

In the previous sections, we analyzed the existing Referrer-Anonymizing Services, we listed some ways that one can legitimately or illegitimately use them and provided empirical data on the types of users that they attract. The final part of the RAS ecosystem are the destination sites (`example.com` in Figure 2.1). It is interesting to know, whether popular web-sites are aware of the existence of RASs and if they are, how do they react towards traffic relayed through them.

In order to identify differences in served content, we conducted an automated experiment involving the top 1,000 sites of the Internet according to Alexa. The first time we visited each Alexa link, we provided the URL of a popular search engine as our referrer header, simulating a user who followed a search result from that search engine. We then repeated the same request 30 times, each time providing as a referrer, the URL of one of our 30 RASs.

Given the dynamic nature of the Web, simply comparing the pages of different visits or their hashes is insufficient to differentiate between changes that were due to the web-site's reaction to a certain referrer header and usual expected changes, such as different timestamps or randomly generated data embedded in the source-code of the page. In order to overcome this obstacle we used a combination of two methods. The first method was to apply Arc90's *Readability* algorithm [10] which attempts to separate and show the most important content on a page while hiding the less important. In the second method, we recorded the number and type of HTML input elements present on the page. The rationale behind the second method was that, even if a page legitimately changes between successive visits, the number of visible and invisible input elements should not change. If any one of the two methods provided different results between the first search-engine-referred visit of the site and any of the the RAS-referred ones, the site and its HTML code was recorded and the results were manually verified.

From a total of 1,000 web-sites we discovered that three of them were using the referrer header to provide radically different content. The first, `facebook.com`, was serving a very different page when our requests claimed to come from one of the 30 studied RASs. By manually checking the resulting page, we realized

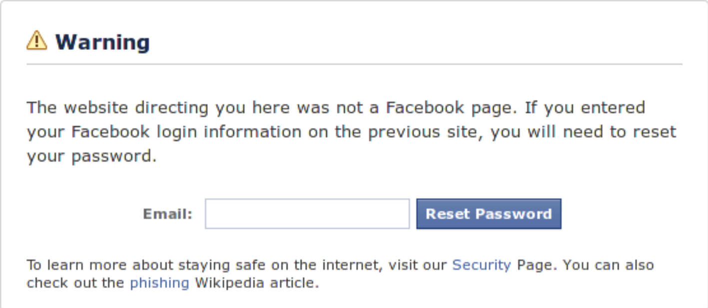

Figure 2.3: Facebook's main page when visited through a specific RAS

that instead of Facebook's usual login-screen, we received a page that alerted us that we had most likely been a victim of a phishing attack, and was inviting us to start the procedure of resetting our password (Figure 2.3). Interestingly, `facebook.com` reacted this way only when the referrer header was coming from that specific RAS and provided the normal content for the remaining 29 RASs. This could mean that Facebook was reacting to a real phishing attack where the attackers were trying to hide the location of the phishing page by redirecting their victims back to `facebook.com` through that specific RAS after the user credentials had been stolen.

The second web-site was a photo gallery where users can upload pictures and retrieve links that they can later use in various web-sites such as forums and blogs. When the site was visited through one of the 30 RASs, instead of giving us access to its main page, it provided us with a GIF image that stated that the site had blocked access to the linking page. This picture will be provided to any image requests that pass through that RAS. This verifies the behavior that we discovered in our own RAS, where visitors were linking to adult content uploaded to generic image galleries and hiding the linking site through the anonymization of their referrer header. The third site was a general information portal which was consistently providing a 403 HTTP 'forbidden' error when visited through a specific RAS, the same RAS blacklisted by the image gallery site.

The above experiment shows that even though only a small fraction of the tested web-sites reacted visibly to specific referrers, their behavior was always 'negative' when the referrer appeared to be a RAS. This attests towards the argument that RASs are associated more with illegal activity and less with a legitimate user's privacy concerns.

2.8 Related Work

Although the practice of cleaning the ‘Referer’ header through a RAS is common knowledge, we are unaware of any research into the operation or usage of these services, with regard to online privacy and anonymity. The related work that we are aware of, falls in the following 4 categories:

Online privacy and anonymity Online privacy and anonymity are important for numerous reasons. The Internet was not built to provide anonymous communication mechanisms, which lead to the creation of various projects that provide forms of anonymous networking. The Onion Router (Tor) [188] project and the Invisible Internet Project (I2P) [66] are the most famous of these networks.

Experiments It is easy to state that a security problem is real and dangerous. Providing evidence backing up this claim is often difficult since it involves covertly tracking the behavior of attackers and victims in an ethical way.

In our experiments in Section 2.5, we used enticing links to lure visitors to our own *fileleaks.co.cc* in a honeypot-like way. Honeypots [143] have been traditionally used to study attacking techniques and post-exploitation trends. Yuil et al. [224] introduce Honeyfiles as an intrusion detection tool to identify attackers. Honeyfiles are bait files that are stored on, and monitored by, a server. These files are intended to be opened by attackers and when they do so, the server emits an alert. Similarly, Bowen et al. [22] use files with ‘stealthy beacons’ to identify an insider thread. We have used these techniques in the past to detect whether attackers are violating the assumed privacy in file-hosting services [122].

Referrer abuse Referrer headers were designed to identify which URL a visitor is coming from. This information is of great benefit to content-providers because it can provide some insight in the browsing habits of visitors, as discussed in Section 2.2.2. The referrer data however, is optional and can be spoofed (e.g. RefSpoof [41]), prompting the inception of *referrer spam* [208]: sending the URL for a third-party web-site in the referrer header so it will show up in the logs of a visited web-site.

Because of their use to track visitor movements, everyone naturally expects referrer headers to contain URLs. This expectation can result in the development of a web-application which displays the contents of the referrer header, without

sufficient sanitization. In such an application, the referrer header can be abused to carry an XSS attack as documented in [48].

Many Cross-site Request Forgery (CSRF) countermeasures depend on the referrer header to determine whether a request was sent from a trusted location. The authors of some of these CSRF countermeasures, aware of visitors that disable the transmission of referrer information, will implement lenient referrer validation [15], which will allow requests without referrer header in order not to break the web-application that is being protected. This deliberate loophole allows an attacker to launch a CSRF attack by relaying a request from an untrusted location through a RAS, which will remove the referrer information. Because the request in this attack has no referrer information, it is allowed by the CSRF countermeasure and the attack can succeed.

Solutions dealing with referrers There are only two parties that can benefit from non-disclosure of referrer information: the visiting browser and the author of the web-site on which a link is hosted. The referrer-leakage problem can thus be solved by either party.

Referrer-Anonymizing Services attempt to solve privacy and anonymity issues that arise because a visitor's browser is leaking information through the referrer header by design. The author of a web-site linking to an external web-page used to not have any other means to prevent the referrer header from exposing their web-site. One way the author of a web-site could prevent the referrer header from exposing their web-site, was to host their web-site using HTTPS. The HTTP protocol specification [151] advises that referrer information should not be sent when navigating from an HTTPS web-site to an HTTP site. However, browsers are free to preserve the referrer if the destination is also an HTTPS web-site, even if the destination site is situated on a different domain.

Recognizing the need for a better solution, the WHATWG has included the 'noreferrer' link type [207] in the HTML5 specification. By annotating certain HTML elements with this link type, a web-page author will prevent referrer information from leaking when clicking the annotated link. RASs protect the web-site's anonymity as much as they protect a visitor's privacy. Therefore it makes sense for an Internet user to disable referrer information to safeguard that privacy at the source. Many modern web-browsers provide means to disable referrer header transmission [110, 20, 134]. For other browsers, the referrer can be filtered out using a client-side proxy like e.g. Privoxy [65]. Due to the privacy problems associated with the referrer header, the 'Origin' header [14] has been proposed because it only leaks the origin (scheme, hostname and port number) of a URL to a remote web-site instead of the full URL.

2.9 Conclusion

In this chapter, we explored the ecosystem of Referrer-Anonymizing Services and classified their functionality, their user-base and their abuse. We showed that in several cases, RASs were taking advantage of their position and leaked private user information to advertising companies. Conversely, we discovered that users were occasionaly using RASs to hide illegal or unethical activity and we revealed that some popular Internet sites do not respond well to RAS-relayed traffic. Overall we showed that, while protecting a user's privacy through the anonymization of the referrer header is desirable, not all RASs are equally noble and thus care should be taken when choosing one. At the same time, browser developers have the responsibility to facilitate a complete migration away from such services through the support of privacy-preserving HTML5 tags.

Chapter 3

File Hosting Services

Preamble

This chapter presents a systematic exploration of File Hosting Services with a focus the privacy and isolation of users' files. The contents of this chapter are replicated from the paper titled "Exposing the Lack of Privacy in File Hosting Services" [122], which was published in the proceedings of the 4th USENIX Workshop on Large-scale Exploits and Emergent Threats (LEET), in 2011. This work was done with the collaboration of other authors from KU Leuven and EURECOM. Nick Nikiforakis was the lead author of this paper.

3.1 Introduction

In an ever expanding Internet, an increasing number of people utilize online services to share digital content. Most of the platforms that support file sharing operate in a broadcasting way. For example, users of social networks can upload multimedia files which are then accessible to their entire friend list. In traditional peer-to-peer networks (e.g., DC++, KaZaa and BitTorrent), users share their files with everyone connected to the same network.

Sometimes however, users may want to share files only with a limited number of people, such as their co-workers, spouses, or family members. In these situations, the all-or-nothing approach of sharing files is not desired. While emailing digital content is still a popular choice, most email providers do not usually allow attachments that exceed a few tens of megabytes.

One of the first services designed to fill this gap was offered by RapidShare[1], a company founded in 2006. RapidShare provides users the ability to upload large files to their servers and then share the links to those files with other users. RapidShare's success spawned hundreds of file hosting services, (a.k.a. one-click hosters), that compete against each other for a share of users. Apart from being used as a way to share private files, researchers have found that FHSs are also used as an alternative to peer-to-peer networks [8], since they offer several advantages such as harder detection of the user who first uploaded a specific file, always-available downloads and no upload/download ratio [2] measurements.

In this chapter, we present our study on 100 file hosting services. These services adopt a security-through-obscurity mechanism where a user can access the uploaded files only by knowing their correct download URIs. While these services claim that these URIs are secret and cannot be guessed, our study shows that this is far from being true. A significant percentage of FHSs generate the "secret" URIs in a predictable fashion, allowing attackers to easily enumerate their files and get access to content that was uploaded by other users.

We implemented crawlers for different hosting providers and, using search engines as a privacy classification mechanism, we were able to disclose hundreds of thousands of private files in less than a month. The second contribution of this chapter is a technique to measure whether attackers are already abusing FHSs to access private information. To reach this goal, we created a number of "HoneyFiles", i.e. fake documents that promise illegal content, and we uploaded them to many FHSs. These files were designed to silently contact one of our servers once they were opened. Over the period of a month, more than 270 file accesses from over 80 different IP addresses were recorded, showing that private documents uploaded to file hosting services are in fact actively downloaded by attackers.

3.2 Life cycle of files on File Hosting Services

In this section, we describe the typical life cycle of a file in relation to file hosting services and we point out the privacy-sensitive steps. In order to model this life cycle we consider a FHS which allows users to upload files without the need to register an account.

A typical user interaction with a FHS usually follows the following steps:

[1] RapidShare AG, `http://www.rapidshare.com/`

[2] A user-quality metric used mainly in private BitTorrent trackers

1. *Alice* decides to share some digital content[3] with other users by uploading it to her favorite FHS.
2. The FHS receives and stores the file. It then creates a *unique identifier* and binds it to the uploaded file.
3. The identifier (ID) is returned to *Alice* in the form of a *URI* that permits her to easily retrieve the uploaded file.
4. *Alice* may now distribute the URI according to the nature of her file: If the file is meant to be public, she may post the link on publicly accessible forums, news-letters or social-networks, otherwise she may prefer to share the URI using one-to-one communication channels such as e-mails or instant messaging.
5. The public or private recipients use the shared URI to access the file that *Alice* has uploaded.
6. If the uploaded file violates copyright laws (movies, music, non-free software) then a third party can possibly report it. In this case the file is deleted and the unique ID is possibly reused. If the file survives this process, it remains accessible for a limited amount of time (set by the provider) or until *Alice* voluntarily removes it.

In the context of this chapter, we focus on the scenario in which the user uploads a *private* file to a FHS and uses a one-to-one channel to share the unique ID returned by the service with the intended recipients of the file. We also assume that the trusted recipients will not forward the URI to other untrusted users.

In this case, the only way in which the file can be accessed by a malicious user is either by guessing the unique ID returned by the FHS or by exploiting a software bug that will eventually allow access to the uploaded files.

3.3 Privacy study

The first phase of our study consists in comparing the privacy offered by 100 File Hosting Services. The list was constructed by merging the information retrieved from different sources, such as the Alexa website, Google search engine, and a number of articles and reviews posted on the Web [111, 169].

Our final top 100 list includes well-known FHSs like `RapidShare`, `FileFactory` and `Easyshare` as well as less popular and regional websites like `FileSave.me` (India), `OnlineDisk.ru` (Russia) and `FileXoom.com` (Mexico).

Before starting the experiments we manually screened each website and found that 12 of them provide either a search functionality (to retrieve the link of a

[3] In the rest of the chapter called “file”

file knowing its name) or an online catalogue (to browse part of the uploaded files). Obviously, these services are not intended for storing personal documents, but only as an easy way to publicly share files between users. Therefore, we removed these FHSs from our privacy study and we focused our testing on the remaining 88 services.

We began our evaluation by analyzing how the unique *file identifiers* are generated by each FHS. As we described in the previous section, when a user uploads a file to a FHS, the server creates an unique identifier (ID) and binds it to the uploaded file. The identifier acts as a shared secret between the user and the hosting service, and therefore, it should not be possible for an attacker to either guess or enumerate valid IDs.

Sequential Identifiers

For each FHS, we consecutively uploaded a number of random files and we monitored the download URIs generated by the hosting service. Surprisingly, we noticed that 34 out of the 88 FHSs (38.6%) generated sequential IDs to identify the uploaded files. Hence, a hypothetical attacker could easily enumerate all private files hosted by a vulnerable FHS by repeatedly decreasing a valid ID (that can be easily obtained by uploading a test file).

As an example, the following snippet shows how one of the hosting providers (anonymized to *vulnerable.com*) stored our uploaded files with sequential identifiers:

```
http://vulnerable.com/9996/
http://vulnerable.com/9997/
http://vulnerable.com/9998/
http://vulnerable.com/9999/
[...]
```

However, the enumeration attack is possible only when the URI does not contain other *non-guessable* parameters. In particular, we noticed that out of the 34 FHS that use sequential identifiers, 14 also required the proper filename to be provided with the secret identifier. For example, the URI was sometimes

	Sequential ID	Non-Sequential ID	Tot
Filename:			
Required	14	6	20
Not required	20	48	68
Total	34	54	88

Table 3.1: Analysis of the Download URI's identifier

constructed by appending the filename as shown in the following example:

```
http://site-one.com/9996/foo.pdf
http://site-one.com/9997/bar.xls
http://site-one.com/9998/password.txt
[...]
```

Since the filenames associated to each ID are, in general, unknown to the attacker, this feature acts as an effective mitigation against enumeration attacks. Even though it would still be possible for an attacker to narrow down his research to a particular filename, we did not investigate this type of targeted attacks in our study.

Table 3.1 provides an overview of the techniques adopted by the various FHSs to generate the download URIs. The privacy provided by 20 service providers was extremely weak, relying only on a sequential ID to protect the users' uploaded data. Unfortunately, the problem is extremely serious since the list of insecure FHSs using sequential IDs also includes some of the most popular names, often highly ranked by Alexa in the list of the top Internet websites.

To further prove that our concerns have a practical security impact, we implemented an automatic enumerator for the 20 FHSs that use sequential IDs and do not require the knowledge of the associated filename. Our tool inserted a random delay after each request to reduce the impact on the performance of the file hosting providers. As a result, our crawler requests were interleaved with many legitimate user requests, a fact which allowed us to conduct our experiment for over a month without being blacklisted by any service provider.

When a user requests a file by providing a valid URI, the FHS usually returns a page containing some information about the document (e.g., filename, size, and number of times it was downloaded), followed by a series of links which a user must follow to download the real file. This feature is extremely convenient for an attacker that can initially scrape the name of each file, and then download only those files that look more interesting.

By enumerating sequential IDs, our crawler was able to retrieve information

File Type	# Enumerated Files
Images (JPG, GIF, BMP)	27,771
Archives (ZIP)	13,354
Portable Document Format (PDF)	7,137
MS Office Word Documents	3,686
MS Office Excel Sheets	1,182
MS PowerPoint	967

Table 3.2: Number of files with sensitive types reported as private by our privacy-classification mechanism

about 310,735 unique files in a period of 30 days. While this list is "per se" sensitive information, we tried to estimate how many files correspond to private users' documents.

It is reasonable to assume that if the user wants to make her file publicly accessible, she would post the download URI on websites, blogs, and/or public forums, depending on the target audience of the file. On the other hand, if a file is intended to be kept private, the link would probably be shared in a one-to-one fashion, e.g., through personal emails and instant messaging. We decided to exploit this difference to roughly characterize a file as *public* or *private*. In particular, we queried for each file name on Bing, Microsoft's search engine, and we flagged as "public" any file whose link was found on the Web. If Bing did not return any results for our query, we considered the file as private.

Out of the 310,735 unique filenames extracted with our enumeration tool, Bing returned no search results for 168,320, thus classifying 54.16% of files as private. This classification is quite approximate as the list of private files also contains data exchanged in closed "pirate" communities, beyond the reach of search engines' crawlers [8, 89]. Nevertheless, this technique still provides a useful estimation of the impact of this kind of attack on the privacy of FHS users.

Table 3.2 shows the number of unique private files crawled by our tool, grouped by common filetypes. In addition to the major extensions reported in the Table, we also found several files ending with a `.sql` extension. These files are probably database dumps that attackers may use to gain a detailed view of the content of the victim's database.

Even though we believe that the majority of these files contain private information, for ethical reasons we decided not to verify our hypothesis. In fact, to preserve the users' privacy, our crawler was designed to extract only the name and size of the files from the information page, without downloading the actual content.

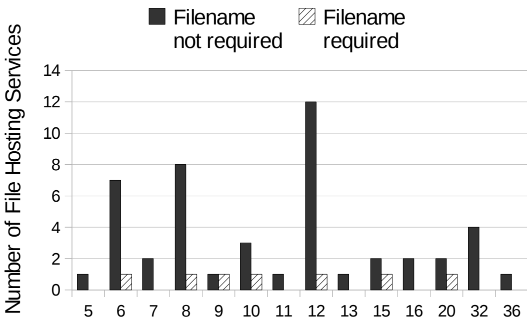

Figure 3.1: Length of the Identifier

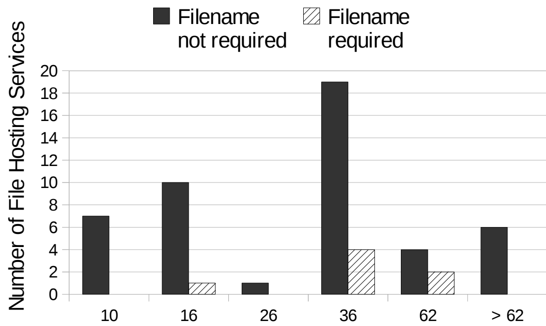

Figure 3.2: Size of the Identifier's Character Set

Random Identifiers

In a second experiment we focused on the 54 FHSs that adopt *non sequential* identifiers (ref. Table 3.1).

In these cases, the identifiers were randomly generated for each uploaded file, forcing a malicious user to guess a valid random value to get access to a single file. The complexity of this operation depends on the length (i.e., number of characters) of the secret and on the character set (i.e., number of possible values for each character) that is used to generate the identifier. As shown in Figure 3.1 and Figure 3.2, different FHSs use different techniques, varying in length from 6 to 36 bytes and adopting different character sets.

The three peaks in Figure 3.1 correspond to identifier length of six, eight, and twelve characters respectively. The second graph shows instead that the most

Length	Chars Set	# Tries	Files Found
6	Numeric	617,169	728
6	Alphanumeric	526,650	586
8	Numeric	920,631	332

Table 3.3: Experiment on the non-sequential identifiers

Feature	Number of FHSs
CAPTCHA	30%
Delay	49%
Password	26%
PRO Version	53%
Automated File Trashing (2-360 days)	54%

Table 3.4: Security features

common approach consists in using alphanumeric characters. Unfortunately, a fairly large number of FHSs are still adopting easily guessable identifiers composed only of decimal or hexadecimal digits.

To show the limited security offered by some of the existing solutions we conducted another simple experiment. We modified our enumerator tool to bruteforce the file identifiers of three different *non sequential* FHSs that did not require the filename in the URI. In particular, we selected a FHS with numeric IDs of 6 digits, one with numeric IDs of 8 digits, and one with alphanumeric IDs of 6 characters. Our tool ran for five days, from a single machine on a single IP address. The results, shown in Table 3.3, confirm that if the random identifiers are too weak, an attacker can easily gain access to thousands of third-party files in a reasonable amount of time.

It is also interesting to note that the success rate of guessing both numeric and alphanumeric IDs of six digits was about the same (1.1 hit every thousand attempts). This shows that the size of the space to explore is not the only variable in the process and that the important factor is the ratio between the number of possible identifiers and the total number of uploaded files.

Existing Mitigations

We have shown that different hosting providers, by adopting download URIs that are either sequential or easily predictable, can be abused by malicious users to access a large amount of private user data. However, some hosting

providers implement a number of mechanisms that mitigate the possible attacks and make automated enumeration more difficult to realize.

From a manual analysis of the FHSs in our list we identified two techniques commonly used to prevent automated downloads. As summarized in Table 3.4, 30% of websites use CAPTCHA and 49% force the user to wait (between 10 to 60 seconds) before the download starts. However, note that both techniques only increase the difficulty of downloading files, and not the process of guessing the right identifier. As a consequence, our tool was not restricted in any way by these protection mechanisms. In addition, we noticed that a large number of sites offer a *PRO version* of their service where these "limitations" are removed by paying a small monthly fee.

A much safer solution consists in adding an online password to protect a file. Unfortunately, as shown in Table 3.4, only a small percentage of the tested FHSs provided this functionality.

Other Design and Implementation Errors

According to the results of our experiments, many FHSs are either vulnerable to sequential enumeration or they generate short identifiers that can be easily guessed by an attacker. In the rest of this section, we show that, even when the identifiers are strong enough to resist a direct attack, other weaknesses or vulnerabilities in the FHS's software may allow an attacker to access or manipulate private files.

While performing our study, we noticed that 13% of hosting services use the same, publicly available, software to provide their services. To verify our concern about the security of these systems, we downloaded and audited the free version of that platform. Through a manual investigation we were able to discover serious design and implementation errors. For instance, the software contained a directory traversal vulnerability that allows an attacker to list the URIs of all the recently uploaded files.

In addition, the software provides to the user a delete URI for each uploaded file. The delete URI can be used by the user at any time to delete her uploaded files from the service's database. This feature can improve the user's privacy since the file can be deleted when it is no longer needed. The deletion ID generated by this software was 14 characters long, with hexadecimal characters. This provides 16^{14} valid combinations which make any attack practically impossible. However, we noticed that the "report file" link, a per file automatically generated link to report copyright violations, consisted of the first 10 characters of the deletion code.

Filename	Claimed Content	% of Access
phished_paypal_details.html	Credentials to PayPal accounts	40.36%
card3rz_reg_details.html	Welcoming text to our fake carding forum and valid credentials	21.81%
Paypal_account_gen.exe	Application which generates PayPal accounts	17.45%
customer_list_2010.html	Leaked customer list from a known law firm	9.09%
Sniffed_email1.doc	Document with an embedded customer list of a known law firm	6.81%
SPAM_list.pdf	List of email addresses for spamming purposes	5.09%

Table 3.5: Set of files containing fake data that were used as bait in our HoneyFiles experiment. The third column shows the resulting download ratio of each file by attackers

Since the "report file" link (used to report a copyright violation) is publicly available to everybody, an attacker can use it to retrieve the first 10 digits of the delete URI, thus lowering the number of combinations to bruteforce to only $16^4 = 65,536$.

To conclude, it is evident that even if end-users only share the download-link with their intended audience, they can not be certain that their files will not reach unintended recipients. In Section 3.5, we will discuss a client-side solution that can protect a user's files without changing his file-uploading and file-sharing habits.

3.4 HoneyFiles

In Section 3.3 we showed that a significant percentage of file hosting services use a URI generation algorithm that produces sequential identifiers, allowing a potential attacker to enumerate all the files uploaded by other users. In addition, our experiments also showed that some of the FHSs which use a more secure random algorithm, often rely on weak identifiers that can easily be bruteforced in a short amount of time.

In summary, a large amount of the top 100 file hosting services are not able to guarantee to the user the privacy of her documents. The next question we investigate is whether (and to what extent) the lack of security of these websites is already exploited by malicious users. In order to answer this question we

designed a new experiment inspired by the work of Bowen et al. [22] and Yuill et al. [224] on the use of decoy documents to identify insider threats and detect unauthorized access.

First, we registered the *card3rz.co.cc* domain and created a fake login page to a supposedly exclusive underground “carding”[4] community. Second, we created a number of decoy documents that promised illegal/stolen data to the people that accessed them. Each file contained a set of fake sensitive data and some text to make the data convincing when necessary. The first two columns of Table 3.5 list the names and descriptions of each file. The most important characteristic of these files is the fact that, once they are open, they automatically connect back to our monitor running on the *card3rz.co.cc* server. This was implemented in different ways, depending on the type of the document. For example, the HTML files included an `<img/>` tag to fetch some content from our webpage, the `exe` file opened a TCP connection upon its execution, and the PDF documents asked the user permission to open a webpage. For Microsoft’s DOC format, the most straightforward way we found was to embed an HTML file inside the document. In cases where user action was required (e.g., the acceptance of a warning dialog or the double-click of the HTML object in the DOC file) we employed social engineering to convince the malicious user to authorize the action.

In addition to the connect-back functionality, one of the files contained valid credentials for logging into our fake carding website. We did this to investigate whether attackers would not only access illegally obtained data but also take action on the information found inside the downloaded files.

The last step of our experiment consisted in writing a number of tools to automatically upload the HoneyFiles to the various FHSs. Since the links to our files were not shared with anyone, any file access recorded by our monitor was the consequence of a malicious user that was able to download and open our HoneyFiles, thus triggering the hidden connect-back functionality.

Monitoring Sequential FHSs

In our first experiment we used our tools to upload the HoneyFiles 4 times a day to all the FHSs adopting sequential identifiers. We also included the ones that have Search/Catalogue functionality in order to find out whether attackers search for illegal content in FHSs.

While we were initially skeptical of whether our experiment would provide positive results, the activity recorded on our monitor quickly proved us wrong.

[4] Carding is a term used for a process to verify the validity of stolen credit card data.

Countries	**Accesses**
Russia	50.06%
Ukraine	24.09%
United States, United Kingdom, Netherlands, Kazakhstan, Germany	2.40% each
Sweden, Moldova, Latvia, India, France, Finland, Egypt, Canada, Belarus, Austria	1.20% each

Table 3.6: Attack geolocation recorded by our HoneyFile monitor

Over the span of one month, users from over 80 unique IP addresses accessed the HoneyFiles we uploaded on 7 different FHSs for a total of 275 times. Table 3.6 shows the categorization of the attackers by their country of origin using geolocation on their IP addresses. While most of the attacks originated from Russia, we also recorded accesses from 16 other countries from Europe, the United States and the Middle East, showing that this attack technique is used globally and it is not confined to a small group of attackers in a single location.

The third column of Table 3.5 presents the download ratio of each HoneyFile. It is evident that attackers favor content that will give them immediate monetary compensation (such as PayPal accounts and credentials for our carding forum) than other data (e.g., email addresses and customer lists). Out of the 7 reported FHSs, one had a catalog functionality (listing all the uploaded files), two of them had a search option and the remaining four had neither catalog nor search functionality. Interestingly, one of the FHSs providing a search functionality did so through a different website, violating its stated Terms of Service (ToS). This shows that apart from abusing sequential identifiers attackers are also searching for sensitive keywords in FHSs that support searching.

Our monitor also recorded 93 successful logins from 43 different IP addresses at our fake carding website using the credentials that we included inside our HoneyFiles. When a valid username and password combination was entered, the carding website informed the user that the website was under maintenance and that she should have retried later. Fourteen out of the 43 attackers did so with the notable example of an attacker that returned to the website and logged in 14 times in a single day. The multiple successful logins show that attackers do not hesitate to make use of the data they find on FHSs. We assume that the fake PayPal credentials were also tried but we have no direct way to confirm our hypothesis.

In addition to login attempts, we also logged several attempts of SQL injection and file inclusion attacks conducted against the login page of our fake carding website and against our monitoring component. This shows that the attackers

who downloaded the files from the vulnerable FHSs had at least some basic knowledge of web hacking techniques and were not plain users that somehow stumbled upon our HoneyFiles. We were also able to locate a post in an underground Russian forum that listed our fake carding website.

Monitoring Non-Sequential FHSs

For completeness, we decided to repeat the HoneyFiles experiment on 20 FHSs that adopt non-sequential identifiers. Interestingly, our monitor recorded 24 file accesses from 13 unique IP addresses originating from decoy documents placed in three separate FHSs over a period of 10 days. Upon examination, two of them were offering search functionality. While the third FHS stated specifically that all files are private and no search option is given, we discovered two websites that advertised as search engines for that FHS. Since our Honeyfiles could be found through these search engines and we never posted our links in any other website, the only logical conclusion is that the owners of that FHS partnered with other companies, directly violating their privacy statement.

We believe that the above experiments show, beyond doubt, that FHSs are actively exploited by attackers who abuse them to access files uploaded by other users. Unfortunately, since FHSs are server-side applications, the users have little-to-no control over the way their data is handled once it has been uploaded to the hosting provider.

3.5 Countermeasures

In previous sections, we showed that not only many file hosting services are insecure and exploitable, but also that they are in fact being exploited by attackers to gain access to files uploaded by other users. This introduces significant privacy risks since the content that users uploaded, and that was meant to be privately shared, is now in the hands of people who can use it for a variety of purposes, ranging from blackmailing and scamming to identity theft.

We notified 25 file hosting providers about the problems we found in our experiments. Some of them already released a patch to their system, for instance by replacing sequential IDs with random values. Others acknowledged the problem but, at the time of writing, they are still in the process of implementing a solution. Unfortunately, not all the vendors reacted in the same way. In one case, the provider refused to adopt random identifiers because it would negatively affect the performance of the database server, while another provider

"solved" the problem by changing the Term of Service (ToS) to state that his system does not guarantee the privacy of the uploaded files.

Therefore, even though it is important to improve the security on the server side, countermeasures must also be applied on the client side to protect the user's privacy even if her files end up in the hands of malicious users.

An effective way of securing information against eavesdroppers is through encryption, for example by using password-protected archives. In some cases, however, the default utilities present in operating systems cannot correctly handle encrypted archive files.[5] Therefore, we decided to design and implement a client-side security mechanism, *SecureFS*, which automatically encrypts/decrypts files upon upload/download and uses steganographic techniques to conceal the encrypted files and to present a fake one to the possible attackers. The motivation behind SecureFS is to transparently protect the user's files as well as providing a platform for detecting attackers and insecure file hosting services in the future.

SecureFS is implemented as a Firefox extension that constantly monitors file uploads and downloads to FHSs through the browser. When SecureFS detects that the user is about to upload a file, it creates an encrypted copy of the document and combines it with a ZIP file containing fake data. Due to the fact that ZIP archives place their metadata at the end of the file, a possible attacker who downloads the protected file will decompress it and access the fake data without realizing that he is being mislead. Even if the attacker notices that something is wrong (e.g., by noticing the size difference between the ZIP file and the resulting file) the user's file is still encrypted and thus protected. On the other hand, when a legitimate user downloads the file, SecureFS recognizes its internal structure and automatically extracts and decrypts the original file. Most of the described process is automatic, transparent and performed without the user's assistance, allowing users to protect their files without changing their file-sharing routine.

3.6 Ethical Considerations

Testing the security of one hundred file hosting providers and extracting information for thousands of user files may raise ethical concerns. However, analogous to the real-world experiments conducted by Jakobsson et al. [68, 69], we believe that realistic experiments are the only way to reliably estimate success rates of attacks in the real world. Moreover, we believe that our experiments

[5] How to open password-protected ZIP in Mac OS X, `http://www.techiecorner.com/833/`

helped some file hosting providers to improve their security. In particular, note that:

- The enumerator tools accessed only partial information of the crawled files (the file's name and size) and did not download any file content.
- The enumerator tools employed a random delay between each requests to avoid possible impacts on the performance of the file hosting providers.
- We did not break into any systems and we immediately informed the security department of the vulnerable sites of the problems we discovered.
- The HoneyFiles were designed to not harm the user's computer in any way. Moreover, we did not distribute these files on public sites, but only uploaded them (as private documents) to the various FHSs.

3.7 Related Work

Different studies have recently been conducted on the security and privacy of online services. For example, Balduzzi et al. [13] and Gilbert et al. [210] analyze the impact of social-networks on the privacy of Internet users, while Narayanan et al. [116] showed that by combining public data with background knowledge, an attacker is capable of revealing the identify of subscribers to online movie rental services. Other studies (e.g., [165, 213]) focused on the security and privacy trends in mass-market ubiquitous devices and cloud-computing providers [139].

However, to the best of our knowledge, no prior studies have been conducted on the privacy of FHSs. In this chapter we reported the insecurity of many hosting providers by experimentally proving that these services are actually exploited by attackers. There are, however, a number of cases where sequential identifiers of various services have been exploited. For example, researchers investigated how session IDs are constructed and in which cases they can be bruteforced [46]. The study is not restricted to IDs stored in cookies, but also analyzes the sequential and non-sequential IDs present inside URIs. Recently, researchers also identified issues with sequential identifiers in cash-back systems [187].

Antoniades et al. [8] notice that the recent increase of FHSs is threatening the dominance of peer-to-peer networks for file sharing. FHSs are found to have better performance, more content and that this content persists for a longer time than on peer-to-peer networks like BitTorrent. Researchers have also used FHSs as a distributed mechanism to store encrypted filecaches that can be used by a collaborating group of people [72].

Honeypots [143] have been traditionally used to study attacking techniques and post-exploitation trends. Yuil et al. [224] introduce Honeyfiles as an intrusion detection tool to identify attackers. Honeyfiles are bait files that are stored on, and monitored by, a server. These files are intended to be opened by attackers and when they do so, the server emits an alert. Similarly, Bowen et al. [22] use files with "stealthy beacons" to identify insider threats.

3.8 Conclusion

In this chapter, we investigated the privacy of 100 file hosting services and discovered that a large percentage of them generate download URIs in a predictable fashion. Specifically, many FHSs are either vulnerable to sequential enumeration or they generate short identifiers that can be easily guessed by an attacker. Using different FHS enumerators that we implemented, we crawled information for more than 310,000 unique files. Using the Bing search engine as a privacy-classification mechanism, we showed that 54% of them were likely private documents since they were not indexed by the search engine. We also conducted a second experiment to demonstrate that attackers are aware of these vulnerabilities and they are already exploiting them to gain access to files uploaded by other users. Finally we presented *SecureFS*, a client-side protection mechanism which is able to protect a user's files when uploaded to insecure FHSs, even if the documents ends up in the possession of attackers.

Chapter 4

Remote JavaScript Inclusions

Preamble

This chapter presents a large scale characterization of remote JavaScript inclusions. Our experiment is the largest inclusion-oriented experiment to date. The contents of this chapter are replicated from the paper titled "You Are What You Include: Large-scale Evaluation of Remote JavaScript Inclusions" [123], which was published in the proceedings of the 19th ACM Conference on Computer and Communications Security (CCS), in 2012. This work was done with the collaboration of other authors from KU Leuven and the University of California, Santa Barbara. Nick Nikiforakis was the lead author of this paper.

4.1 Introduction

The web has evolved from static web pages to web applications that dynamically render interactive content tailored to their users. The vast majority of these web applications, such as Facebook and Reddit, also rely on client-side languages to deliver this interactivity. JavaScript has emerged as the de facto standard client-side language, and it is supported by every modern browser.

Modern web applications use JavaScript to extend functionality and enrich user experience. These improvements include tracking statistics (e.g., Google Analytics), interface enhancements (e.g., jQuery), and social integration (e.g., Facebook Connect). Developers can include these external libraries in their

web applications in two ways: either (1) by downloading a copy of the library from a third-party vendor and uploading it to their own web server, or (2) by instructing the users' browsers to fetch the code directly from a server operated by a third party (usually the vendor). The safest choice is the former, because the developer has complete control over the code that is served to the users' browsers and can inspect it to verify its proper functionality. However, this choice comes with a higher maintenance cost, as the library must be updated manually. Another downside is that by not including remote code from popular Content Distribution Networks, the developer forces the users' browsers to download scripts from his own servers even if they are identical with scripts that are already available in the browsers' cache. Moreover, this method is ineffective when the library loads additional, remotely-hosted, code at run time (e.g., like Google Analytics does). A developer might avoid these drawbacks by choosing the second option, but this comes at the cost of trusting the provider of the code. In particular, the provider has complete control over the content that is served to the user of the web application. For example, a malicious or compromised provider might deface the site or steal the user's credentials through DOM manipulation or by accessing the application's cookies. This makes the provider of the library an interesting target for cyber-criminals: after compromising the provider, attackers can exploit the trust that the web application is granting to the provider's code to obtain some control over the web application, which might be harder to attack directly. For example, on the 8th of December 2011 the domain distributing qTip2, a popular jQuery plugin, was compromised [144] through a WordPress vulnerability. The qTip2 library was modified, and the malicious version was distributed for 33 days.

It is generally known that developers should include JavaScript only from trustworthy vendors, though it is frightening to imagine the damage attackers could do when compromising a JavaScript vendor such as Google or Facebook. However, there has been no large-scale, in-depth study of how well the most popular web applications implement this policy. In this chapter, we study this problem for the 10,000 most popular web sites and web applications (according to Alexa), outlining the trust relationships between these domains and their JavaScript code providers. We assess the maintenance-quality of each provider, i.e., how easy it would be for a determined attacker to compromise the trusted remote host due to its poor security-related maintenance, and we identify weak links that might be targeted to compromise these top domains. We also identify new types of vulnerabilities. The most notable is called "Typosquatting Cross-site Scripting" (TXSS), which occurs when a developer mistypes the address of a library inclusion, allowing an attacker to register the mistyped domain and easily compromise the script-including site. We found several popular domains that are vulnerable to this attack. To demonstrate the impact of this attack, we registered some domain names on which popular sites incorrectly bestowed

trust, and recorded the number of users that were exposed to this attack.

The main contributions of this chapter are the following:

- We present a detailed analysis of the trust relationships of the top 10,000 Internet domains and their remote JavaScript code providers.
- We evaluate the security perimeter of top Internet domains that include code from third-party providers.
- We identify four new attack vectors to which several high traffic web sites are currently vulnerable.
- We study how the top domains have changed their inclusions over the last decade.

4.2 Data Collection

In this section, we describe the setup and results of our large-scale crawling experiment of the Alexa top 10,000 web sites.

4.2.1 Discovering remote JavaScript inclusions

We performed a large web crawl in order to gather a large data set of web sites and the remote scripts that they include. Starting with Alexa's list of the top 10,000 Internet web sites [5], we requested and analyzed up to 500 pages from each site. Each set of pages was obtained by querying the Bing search engine for popular pages within each domain. For instance, the search for "site:google.com" will return pages hosted on Google's main domain as well as subdomains. In total, our crawler visited over 3,300,000 pages of top web sites in search for remote JavaScript inclusions. The set of visited pages was smaller than five million since a portion of sites had less than 500 different crawlable pages.

From our preliminary experiments, we realized that simply requesting each page with a simple command-line tool that performs an HTTP request was not sufficient, since in-line JavaScript code can be used to create new, possibly remote, script inclusions. For example, in the following piece of code, the inline JavaScript will create, upon execution, a new remote script inclusion for the popular Google-Analytics JavaScript file:

Offered service	**JavaScript file**	**% Top Alexa**
Web analytics	www.google-analytics.com/ga.js	68.37%
Dynamic Ads	pagead2.googlesyndication.com/pagead/show_ads.js	23.87%
Web analytics	www.google-analytics.com/urchin.js	17.32%
Social Networking	connect.facebook.net/en_us/all.js	16.82%
Social Networking	platform.twitter.com/widgets.js	13.87%
Social Networking & Web analytics	s7.addthis.com/js/250/addthis_widget.js	12.68%
Web analytics & Tracking	edge.quantserve.com/quant.js	11.98%
Market Research	b.scorecardresearch.com/beacon.js	10.45%
Google Helper Functions	www.google.com/jsapi	10.14%
Web analytics	ssl.google-analytics.com/ga.js	10.12%

Table 4.1: The ten most popular remotely-included files by the Alexa top 10,000 Internet web-sites

```
var ishttps = "https:" == document.location.protocol;
var gaJsHost = (ishttps)?
     "https://ssl." : "http://www.");
var rscript = "";
rscript += "\%3Cscript src='" + gaJsHost;
rscript += "google-analytics.com/ga.js' type=";
rscript += "'text/javascript'\%3E\%3C/script\%3E";

document.write(unescape(rscript));
```

To account for dynamically generated scripts, we crawled each page utilizing HtmlUnit, a headless browser [1], which in our experiments pretended to be Mozilla Firefox 3.6. This approach allowed us to fully execute the inline JavaScript code of each page, and thus accurately process all remote script inclusion requests, exactly as they would be processed by a normal Web browser. At the same time, if any of the visited pages, included more remote scripts based on specific non-Firefox user-agents, these inclusions would be missed by our crawler. While in our experiments we did not account for such behaviour, such a crawler could be implemented either by fetching and executing each page with multiple user-agents and JavaScript environments, or using a system like Rozzle [84] which explores multiple execution paths within a single execution in order to uncover environment-specific malware.

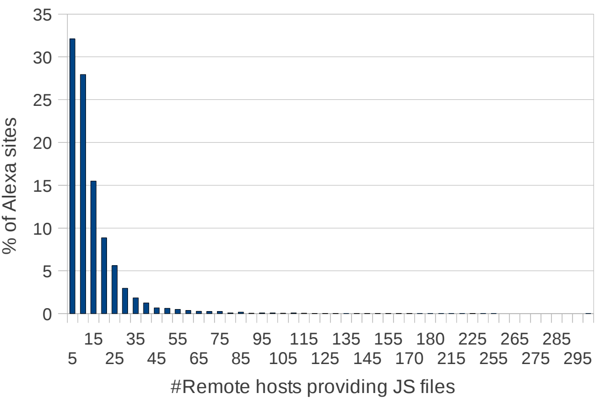

Figure 4.1: Relative frequency distribution of the percentage of top Alexa sites and the number of unique remote hosts from which they request JavaScript code

4.2.2 Crawling Results

Number of remote inclusions

The results of our large-scale crawling of the top 10,000 Internet web sites are the following: From 3,300,000 pages, we extracted 8,439,799 inclusions. These inclusions map to 301,968 unique URLs of remote JavaScript files. This number does not include requests for external JavaScript files located on the same domain as the page requesting them. 88.45% of the Alexa top 10,000 web sites included at least one remote JavaScript library. The inclusions were requesting JavaScript from a total of 20,225 uniquely-addressed remote hosts (fully qualified domain names and IP addresses), with an average of 417 inclusions per remote host. Figure 4.1 shows the number of unique remote hosts that the top Internet sites trust for remote script inclusions. While the majority of sites trusts only a small number of remote hosts, the long-tailed graph shows that there are sites in the top Alexa list that trust up to 295 remote hosts. Since a single compromised remote host is sufficient for the injection of malicious JavaScript code, the fact that some popular sites trust hundreds of different remote servers for JavaScript is worrisome.

[1] HtmlUnit-`http://htmlunit.sourceforge.net`

Remote IP address Inclusions

From the total of 8,439,799 inclusions, we discovered that 23,063 (0.27%) were requests for a JavaScript script, where the URL did not contain a domain name but directly a remote IP address. These requests were addressing a total of 324 unique IP addresses. The number of requesting domains was 299 (2.99% percent of the Alexa top 10,000) revealing that the practice of addressing a remote host by its IP address is not widespread among popular Internet sites.

By geolocating the set of unique IP addresses, we discovered that they were located in 35 different countries. The country with most of these IP addresses is China (35.18%). In addition, by geolocating each domain that included JavaScript from a remote IP address, we recorded only 65 unique cases of cross-country inclusions, where the JavaScript provider and the web application were situated in different countries. This shows that if a web application requests a script directly from a remote host through its IP address, the remote host will most likely be in the same country as itself.

In general, IP-address-based script inclusion can be problematic if the IP addresses of the remote hosts are not statically allocated, forcing the script-including pages to keep track of the remote servers and constantly update their links instead of relying on the DNS protocol.

Popular JavaScript libraries

Table 4.1 presents the ten most included remote JavaScript files along with the services offered by each script and the percentage of the top 10,000 Alexa sites that utilize them. There are several observations that can be made based on this data. First, by grouping JavaScript inclusions by the party that benefits from them, one can observe that 60% of the top JavaScript inclusions do not directly benefit the user. These are JavaScript libraries that offer Web analytics, Market Research, User tracking and Dynamic Ads, none of which has any observable effect in a page's useful content. Inclusions that obviously benefit the user are the ones incorporating social-networking functionality.

At the same time, it is evident that a single company, Google, is responsible for half of the top remotely-included JavaScript files of the Internet. While a complete compromise of this company is improbable, history has shown that it is not impossible [227].

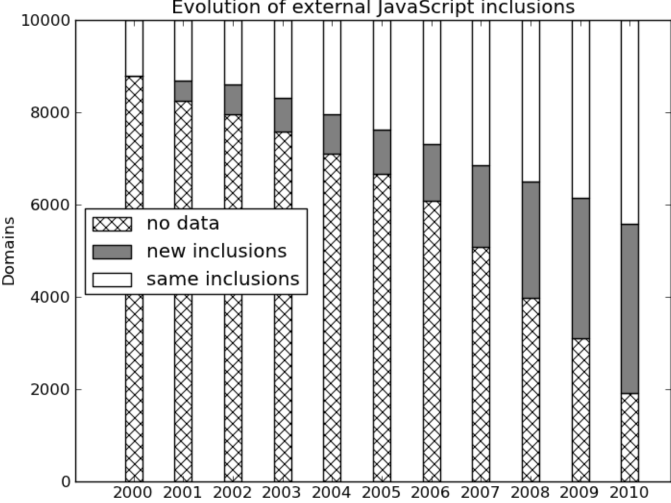

Figure 4.2: Evolution of remote JavaScript inclusions for domains ranked in the top 10,000 from Alexa.

4.3 Characterization of JavaScript Providers and Includers

In this section, we show how the problem of remote JavaScript library inclusion is widespread and underplayed, even by the most popular web applications. First, we observe how the remote inclusions of top Internet sites change over time, seeking to understand whether these sites become more or less exposed to a potential attack that leverages this problem. Then, we study how well library providers are maintaining their hosts, inquiring whether the developers of popular web applications prefer to include JavaScript libraries from well-maintained providers, which should have a lower chance of being compromised, or whether they are not concerned about this issue.

4.3.1 Evolution of remote JavaScript Inclusions

In the previous section, we examined how popular web sites depend on remote JavaScript resources to enrich their functionality. In this section, we examine the remote JavaScript inclusions from the same web sites in another dimension: time. We have crawled *archive.org* [202] to study how JavaScript inclusions

Year	No data	Same inclusions	New inclusions	% New inclusions
2001	8,256	1,317	427	24.48%
2002	7,952	1,397	651	31.79%
2003	7,576	1,687	737	30.40%
2004	7,100	2,037	863	29.76%
2005	6,672	2,367	961	28.88%
2006	6,073	2,679	1,248	31.78%
2007	5,074	3,136	1,790	36.34%
2008	3,977	3,491	2,532	42.04%
2009	3,111	3,855	3,034	44.04%
2010	1,920	4,407	3,673	45.46%

Table 4.2: Evolution of the number of domains with same and new remote JavaScript inclusions for the Alexa top 10,000

Year	Unique domains	Total remote inclusions	Average # of new domains
2001	428	1,447	1.41
2002	680	2,392	1.57
2003	759	2,732	1.67
2004	894	3,258	1.67
2005	941	3,576	1.64
2006	974	3,943	1.61
2007	1,168	5,765	1.67
2008	1,513	8,816	1.75
2009	1,728	11,439	1.86
2010	2,249	16,901	2.10

Table 4.3: Number of new domains that are introduced every year in remote inclusions.

have evolved through time in terms of new remote dependencies and if these increase or decrease over time.

To better understand how JavaScript is included and how the inclusions change over time, we examine each page from different snapshots that span across several years. For the same pages that we crawled in Section 4.2, we have queried *archive.org* to obtain their versions for past years (if available). For each domain, we choose one representative page that has the most remote inclusions

and the highest availability since 2000. For every chosen page we downloaded one snapshot per year from 2000 to 2010. Every snapshot was compared with the previous one in order to compute the inclusion changes.

In Figure 4.2, one can see the evolution of remote JavaScript inclusions for domains ranked in the top 10,000 from Alexa. For every year, we show how the inclusions from the previous available snapshot changed with the addition of new inclusions or if they remained the same. A *new inclusion* means that the examined domain introduced at least one new remote script inclusion since the last year. If the page's inclusions were the same as the previous year, we consider those as *same inclusion*. Unfortunately, *archive.org* does not cover all the pages we examined completely, and thus we have cases where *no data* could be retrieved for a specific domain for all of the requested years. Also, many popular web sites did not exist 10 years ago. There were 892 domains for which we did not find a single URL that we previously crawled in *archive.org*. A domain might not be found on *archive.org* because of one of the following reasons: the website restricts crawling from its robots.txt file (182 domains), the domain was never chosen to be crawled (320 domains) or the domain was crawled, but not the specific pages that we chose during our first crawl (390 domains). In Table 4.2, we show how many domains introduced new inclusions in absolute numbers. In our experiment, we find (not surprisingly) that as we get closer in time to the present, *archive.org* has available versions for more of the URLs that we query for and thus we can examine more inclusions. We discovered that every year, a significant amount of inclusions change. Every year there are additional URLs involved in the inclusions of a website compared to the previous years and there is a clear trend of including even more. Back in 2001, 24.48% of the studied domains had at least one new remote inclusion. But as the web evolves and becomes more dynamic, more web sites extend their functionality by including more JavaScript code. In 2010, 45.46% of the examined web sites introduced a new JavaScript inclusion since the last year. This means that almost half of the top 10,000 Alexa domains had at least one new remote JavaScript inclusion in 2010, when compared to 2009.

But introducing a new JavaScript inclusion does not automatically translate to a new dependency from a remote provider. In Table 4.3, we examine whether more inclusions translate to more top-level remote domains. We calculate the unique domains involved in the inclusions and the total number of remote inclusions. For every page examined, we keep the unique domains involved in its new inclusions, and we provide the average of that number for all available pages per year. There is a clear trend in Table 4.3 that more inclusions result into more external dependencies from new domains. In fact in 2010 we observed that on average each page expanded their inclusions by including JavaScript from 2.1 new domains on average compared to 2009. This trend shows that

the circle of trust for each page is expanding every year and that the surface of attack against them increases.

4.3.2 Quality of Maintenance Metric

Whenever developers of a web application decide to include a library from a third-party provider, they allow the latter to execute code with the same level of privilege as their own code. Effectively, they are adding the third-party host to the *security perimeter* of the web application, that is the set of the hosts whose exploitation leads to controlling the code running on that web application. Attacking the third-party, and then using that foothold to compromise the web application, might be easier than a direct attack of the latter. The aforementioned incident of the malicious modification of the qTip2 plugin [144], shows that cybercriminals are aware of this and have already used indirect exploitation to infect more hosts and hosts with more secure perimeters.

To better understand how many web applications are exposed to this kind of indirect attack, we aim to identify third-party providers that could be a weak link in the security of popular web applications. To do so, we design a metric that evaluates how well a website is being maintained, and apply it to the web applications running on the hosts of library providers (that is co-located with the JavaScript library that is being remotely included). We indicate the low-scoring as potential weak links, on the assumption that unkempt websites seem easier targets to attackers, and therefore are attacked more often.

Note that this metric aims at characterizing how well websites are maintained, and how security-conscious are their developers and administrators. It is not meant to investigate if a URL could lead to malicious content (a la Google Safebrowsing, for example). Also, we designed this metric to look for the signs of low maintenance that an attacker, scouting for the weakest host to attack, might look for. We recognize that a white-box approach, where we have access to the host under scrutiny, would provide a much more precise metric, but this would require a level of access that attackers usually do not have. We identified the closest prior work in establishing such a metric in SSL Labs's SSL/TLS survey [176] and have included their findings in our metric.

Our *Quality of Maintenance* (QoM) metric is based on the following features:

- **Availability:** If the host has a DNS record associated with it, we check that its registration is not expired. Also, we resolve the host's IP address, and we verify that it is not in the ranges reserved for private networks (e.g., `192.168.0.0/16`). Both of these features are critical, because an attacker could impersonate a domain by either registering the domain

name or claiming its IP address. By impersonating a domain, an attacker gains the trust of any web application that includes code hosted on the domain.

- **Cookies:** We check the presence of at least one cookie set as `HttpOnly` and, if SSL/TLS is available, at least one cookie set as `Secure`. Also, we check that at least one cookie has its `Path` and `Expiration` attributes set. All these attributes improve the privacy of session cookies, so they are a good indication that the domain administrators are concerned about security.

- **Anti-XSS and Anti-Clickjacking protocols:** We check for the presence of the `X-XSS-Protection` protocol, which was introduced with Internet Explorer 8 [159] to prevent some categories of Cross-site Scripting (XSS) attacks [135]. Also, we check for the presence of Mozilla's Content Security Policy protocol, which prevents some XSS and Clickjacking attacks [12] in Firefox. Finally, we check for the presence of the `X-Frame-Options` protocol, which aims at preventing ClickJacking attacks and is supported by all major browsers.

- **Cache control:** If SSL/TLS is present, we check if some content is served with the headers `Cache-Control: private` and `Pragma:no-cache`. These headers indicate that the content is sensitive and should not be cached by the browser, so that local attacks are prevented.

- **SSL/TLS implementation:** For a thorough evaluation of the SSL/TLS implementation, we rely on the study conducted by SSL Labs in April 2011. In particular, we check that the domain's certificate is valid (unrevoked, current, unexpired, and matches the domain name) and that it is trusted by all major browsers. Also, we verify that current protocols (e.g, TLS 1.2, SSL 3.0) are implemented, that older ones (e.g., SSL 2.0) are not used, and if the protocols allow weak ciphers. In addition, we check if the implementation is PCI-DSS compliant [138], which is a security standard to which organizations that handle credit card information must comply, and adherence to it is certified yearly by the Payment Card Industry. Also, we check if the domain is vulnerable to the SSL insecure-renegotiation attack. We check if the key is weak due to a small key size, or the Debian OpenSSL flaw. Finally, we check if the site offers Strict Transport Security, which forces a browser to use secure connections only, like HTTPS.

  SSL Labs collected the features described in the previous paragraph nine months before we collected all the remaining features. We believe that this is acceptable, as certificates usually have a lifespan of a few years, and the Payment Card Industry checks PCI-DSS compliance yearly. Also,

Web server	Up-to-date version(s)
Apache	1.3.42, 2.0.65, 2.2.22
NGINX	1.1.10, 1.0.9, 0.8.55, 0.7.69, 0.6.39, 0.5.38
IIS	7.5, 7.0
Lighttpd	1.5 , 1.4.29
Zeus	4.3
Cherokee	1.2
CWS	3.0
LiteSpeed	4.1.3
0w	0.8d

Table 4.4: Up-to-date versions of popular web servers, at the time of our experiment

since these features have been collected in the same period for all the hosts, they do not give unfair advantages to some of them.

- **Outdated web servers:** Attackers can exploit known vulnerabilities in web servers to execute arbitrary code or access sensitive configuration files. For this reason, an obsolete web server is a weak link in the security of a domain. To establish which server versions (in the HTTP `Server` header) should be considered obsolete, we collected these HTTP Server header strings during our crawl and, after clustering them, we selected the most popular web servers and their versions. Consulting change-logs and CVE reports, we compiled a list of stable and up-to-date versions, which is shown in Table 4.4. While it is technically possible for a web server to report an arbitrary version number, we assume that if the version is modified it will be modified to pretend that the web server is more up-to-date rather than less, since the latter would attract more attacks. This feature is not consulted in the cases where a web server does not send a Server header or specifies it in a generic way (e.g., “Apache”).

The next step in building our QoM metric is to weigh these features. We cannot approach this problem from a supervised learning angle because we have no training set: We are not aware of any study that quantifies the QoM of domains on a large scale. Thus, while an automated approach through supervised learning would have been more precise, we had to assign the weights manually. Even so, we can verify that our QoM metric is realistic. To do so, we evaluated with our metric the websites in the following four datasets of domains in the Alexa Top $10,000$:

- **XSSed domains:** This dataset contains 1,702 domains that have been exploited through cross-site scripting in the past. That is, an attacker injected malicious JavaScript on at least one page of each domain. Using an XSS exploit, an attacker can steal the cookies or password as it is typed into a login form [135]. Recently, the Apache Foundation disclosed that their servers were attacked via an XSS vulnerability, and the attacker obtained administrative access to several servers [217]. To build this dataset, we used XSSed [218], a publicly available database of over $45,000$ reported XSS attacks.
- **Defaced domains:** This dataset contains 888 domains that have been defaced in the past. That is, an attacker changed the content of one or more pages on the domain. To build this dataset, we employed the Zone-H database [229]. This database contains more than six million reports of defacements, however, only 888 out of the 10,000 top Alexa domains have suffered a defacement.
- **Bank domains:** This dataset contains 141 domains belonging to banking institutions (online and brick and mortar) in the US.
- **Random domains:** This dataset contains 4,500 domains, randomly picked, that do not belong to the previous categories.

The cumulative distribution function of the metric on these datasets is shown in Figure 4.3. At score 60, we have 506 defaced domains, 698 XSSed domains, 765 domains belonging to the random set, and only 5 banks. At score 120, we have all the defaced and XSSed domains, 4,409 domains from the random set, and all but 5 of the banking sites. The maximum score recorded is 160, held by `paypal.com`. According to the metric, sites that have been defaced or XSSed in the past appear to be maintained less than our dataset of random domains. On the other hand, the majority of banking institutions are very concerned with the maintenance of their domains. These findings are reasonable, and empirically demonstrate that our metric is a good indicator of the quality of maintenance of a particular host. This is especially valid also because we will use this metric to classify hosts into three wide categories: high maintenance (metric greater than 150), medium, and low maintenance (metric lower than 70).

4.3.3 Risk of Including Third-Party Providers

We applied our QoM metric to the top 10,000 domains in Alexa and the domains providing their JavaScript inclusions. The top-ranking domain is `paypal.com`, which has also always been very concerned with security (e.g., it was one

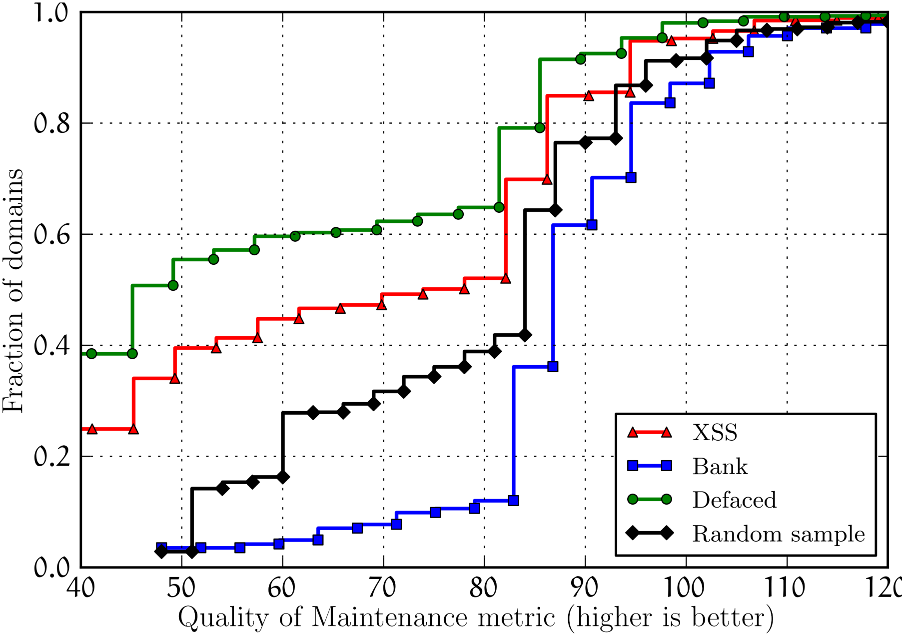

Figure 4.3: Cumulative distribution function of the maintenance metric, for different datasets

of the proposers of HTTP Strict Transport Security). The worst score goes to `cafemom.com`, because its SSL certificate is not valid for that domain (its `CommonName` is set to `mom.com`), and it is setting cookies non-`HTTPOnly`, and not `Secure`. Interestingly, it is possible to login to the site both in HTTPS, and in plain-text HTTP.

In Figure 4.4, we show the cumulative distribution function for the inclusions we recorded. We can see that low-maintenance domains often include JavaScript libraries from low-maintenance providers. High-maintenance domains, instead, tend to prefer high-maintenance providers, showing that they are indeed concerned about the providers they include. For instance, we can see that the JavaScript libraries provided by sites with the worst maintenance scores, are included by over 60% of the population of low-maintenance sites, versus less than 12% of the population of sites with high-maintenance scores. While this percentage is five times smaller than the one of low-maintenance sites, still, about one out of four of their inclusions come from providers with a low maintenance score, which are potential "'weak spots"' in their security perimeter. For example, `criteo.com` is an advertising platform that is remotely included

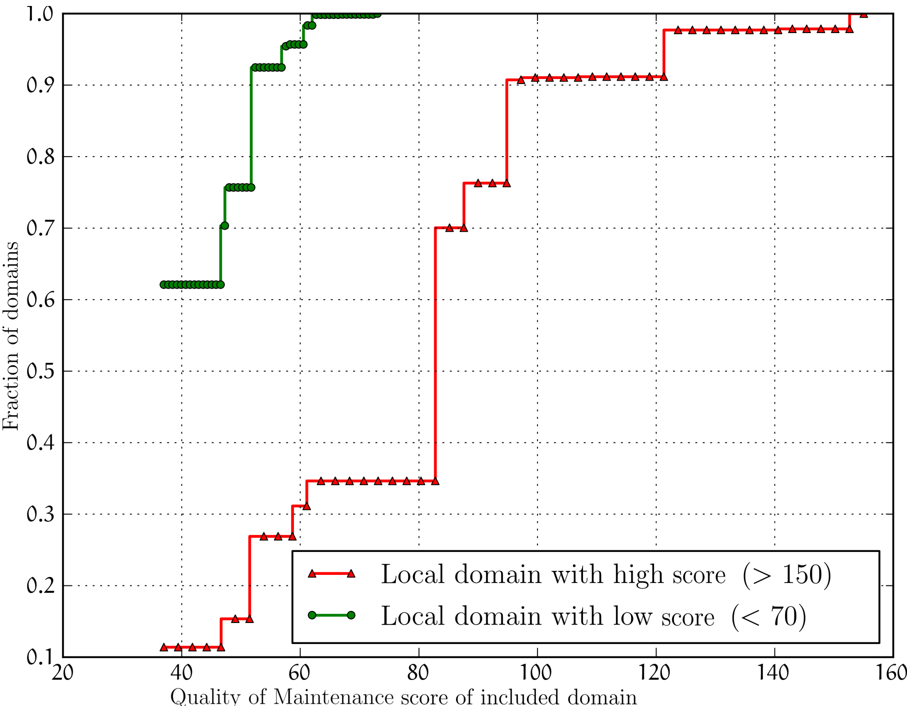

Figure 4.4: Risk of including third-party providers, included in high and low maintenance web applications.

in 117 of the top 10,000 Alexa domains, including `ebay.de` and `sisal.it`, the society that holds the state monopoly on bets and lottery in Italy. `criteo.com` has an implementation of SSL that supports weak ciphers, and a weak Diffie-Hellman ephemeral key exchange of 512 bits. Another example is `levexis.com`, a marketing platform, which is included in 15 of the top 10,000 Alexa websites, including `lastminute.com`, and has an invalid SSL certificate.

4.4 Attacks

In this section, we describe four types of vulnerabilities that are related to unsafe third-party inclusion practices, which we encountered in the analysis of the top 10,000 Alexa sites. Given the right conditions, these vulnerabilities enable an attacker to take over popular web sites and web applications.

4.4.1 Cross-user and Cross-network Scripting

In the set of remote script inclusions resulting from our large-scale crawling experiment, we discovered 133 script inclusions where the “src” attribute of the script tag was requesting a JavaScript file from `localhost` or from the `127.0.0.1` IP address. Since JavaScript is a client-side language, when a user’s browser encounters such a script tag, it will request the JavaScript file from the user’s machine. Interestingly, 131 out of the 133 localhost inclusions specified a port (e.g., `localhost:12345`), which was always greater than 1024 (i.e., a non-privileged port number). This means that, in a multiuser environment, a malicious user can set up a web server, let it listen to high port numbers, and serve malicious JavaScript whenever a script is requested from `localhost`. The high port number is important because it allows a user to attack other users without requiring administrator-level privileges.

In addition to connections to `localhost`, we found several instances where the source of a script tag was pointing to a private IP address (e.g., `192.168.2.2`). If a user visits a site with such a script inclusion, then her browser will search for the JavaScript file on the user’s local network. If an attacker manages to get the referenced IP address assigned to his machine, he will be able to serve malicious JavaScript to the victim user.

We believe that both vulnerabilities result from a developer’s erroneous understanding of the way in which JavaScript is fetched and executed. The error introduced is not immediately apparent because, often times, these scripts are developed and tested on the developer’s local machine (or network), which also hosts the web server.

The set of domains hosting pages vulnerable to cross-user and cross-network scripting, included popular domains such as `virginmobileusa.com`, `akamai.com`, `callofduty.com` and `gc.ca`.

4.4.2 Stale Domain-name-based Inclusions

Whenever a domain name expires, its owner may choose not to renew it without necessarily broadcasting this decision to the site’s user-base. This becomes problematic when such a site is providing remote JavaScript scripts to sites registered under different domains. If the administrators of the including sites do not routinely check their sites for errors, they will not realize that the script-providing site stopped responding. We call these inclusions “stale inclusions”. Stale inclusions are a security vulnerability for a site, since an attacker can register the newly-available domain and start providing all stale JavaScript

inclusion requests with malicious JavaScript. Since the vulnerable pages already contain the stale script inclusions, an attacker does not need to interact with the victims or convince them to visit a specific page, making the attack equivalent to a stored XSS.

To quantify the existence of stale JavaScript inclusions, we first compiled a list of all JavaScript-providing domains that were discovered through our large-scale crawling experiment. From that list, we first excluded all domains that were part of Alexa's top one million web sites list. The remaining 4,225 domains were queried for their IP address and the ones that did not resolve to an address were recorded. The recorded ones were then queried in an online WHOIS database. When results for a domain were not available, we attempted to register it on a popular domain-name registrar.

The final result of this process was the identification of 56 domain names, used for inclusion in 47 of the top 10,000 Internet web sites, that were, at the time of our experiments, available for registration. By manually reviewing these 56 domain names, we realized that in 6 cases, the developers mistyped the JavaScript-providing domain. These form an interesting security issue, which we consider separately in Section 4.4.4.

Attackers could register these domains to steal credentials or to serve malware to a large number of users, exploiting the trust that the target web application puts in the hijacked domain. To demonstrate how easy and effective this attack is, we registered two domains that appear as stale inclusions in popular web sites, and make them resolve to our server. We recorded the `Referer`, source IP address, and requested URL for every HTTP request received for 15 days. We minimized the inconvenience that our study might have caused by always replying to HTTP requests with a HTML-only `404 Not Found` error page, with a brief explanation of our experiment and how to contact us. Since our interaction with the users is limited to logging the three aforementioned pieces of data, we believe there are no ethical implications in this experiment. In particular, we registered `blogtools.us`, a domain included on `goldprice.org`, which is a web application that monitors the price of gold and that ranks 4,779th in the US (according to Alexa). Previously, `blogtools.us` was part of a platform to create RSS feeds. We also registered `hbotapadmin.com`, included in a low-traffic page on `hbo.com`, which is an American cable television network, ranking 1,411th in the US. `hbotapadmin.com` was once owned by the same company, and its registration expired in July 2010. The results of our experiment are shown in Table 4.5. While `hbotapadmin.com` is being included exclusively by HBO-owned domains, it is interesting to notice that `blogtools.us` is still included by several lower-ranking domains, such as `happysurfer.com`, even though the service is not available anymore.

	blogtools.us	hbotapadmin.com
Visits	80,466	4,615
Including domains	24	4
Including pages	84	41

Table 4.5: Results from our experiment on expired remotely-included domains

4.4.3 Stale IP-address-based Inclusions

As described in Section 4.2, some administrators choose to include remote scripts by addressing the remote hosts, not through a domain name but directly through an IP address. While at first this decision seems suboptimal, it is as safe as a domain-name-based inclusion, as long as the IP address of the remote machine is static or the including page is automatically updated whenever the IP address of the remote server changes.

To assess whether one of these two conditions hold, we manually visited all 299 pages performing an IP address-based inclusion, three months after our initial crawl. In the majority of cases, we recorded one of the following three scenarios: a) the same scripts were included, but the host was now addressed through a domain name, b) the IP addresses had changed or the inclusions were removed or c) the IP addresses remained static. Unfortunately, in the last category, we found a total of 39 IP addresses (13.04%) that had not changed since our original crawl but at the same time, were not providing any JavaScript files to the requests. Even worse, for 35 of them (89.74%) we recorded a "Connection Timeout," attesting to the fact that there was not even a Web server available on the remote hosts. This fact reveals that the remote host providing the original scripts either became unavailable or changed its IP address, without an equivalent change in the including pages.

As in domain-name-based stale inclusions, these inclusions can be exploited by an attacker who manages to obtain the appropriate IP address. While this is definitely harder than registering a domain-name, it is still a vulnerability that could be exploited given an appropriate network configuration and possibly the use of the address as part of a DHCP address pool.

4.4.4 Typosquatting Cross-site Scripting (TXSS)

Typosquatting [106, 200] is the practice of registering domain names that are slight variations of the domains associated with popular web sites. For instance, an individual could register `wikiepdia.org` with the intent of capturing a part

Intended domain	Actual domain
googlesyndication.com	googlesyndicatio.com
purdue.edu	purude.edu
worldofwarcraft.com	worldofwaircraft.com
lesechos.fr	lessechos.fr
onegrp.com	onegrp.nl

Table 4.6: Examples of mistyped domains found in remote JavaScript inclusion tags

of the traffic originally meant to go toward the popular Wikipedia website. The user that mistypes Wikipedia, instead of getting a "Server not found" error, will now get a page that is under the control of the owner of the mistyped domain. The resulting page could be used for advertising, brand wars, phishing credentials, or triggering a drive-by download exploit against a vulnerable browser.

Traditionally, typosquatting always refers to a user mistyping a URL in her browser's address bar. However, web developers are also humans and can thus mistype a URL when typing it into their HTML pages or JavaScript code. Unfortunately, the damage of these mistakes is much greater than in the previous case, since every user visiting the page containing the typo will be exposed to data originating from the mistyped domain. In Table 4.6, we provide five examples of mistyped URLs found during our experiment for which we could identify the intended domain.

As in the case of stale domain-names, an attacker can simply register these sites and provide malicious JavaScript to all unintended requests. We observed this attack in the wild: according to Google's Safe Browsing, `worldofwaircraft.com` has spread malware in January 2012. To prove the efficacy of this attack, we registered `googlesyndicatio.com` (mistyped `googlesyndication.com`), and logged the incoming traffic. We found this domain because it is included in `leonardo.it`, an Italian online newspaper (Alexa global rank: 1,883, Italian rank: 56). Over the course of 15 days, we recorded 163,188 unique visitors. Interestingly, we discovered that this misspelling is widespread: we had visitors incoming from 1,185 different domains, for a total of 21,830 pages including this domain. 552 of the domains that include ours belong to blogs hosted on `*.blogspot.com.br`, and come from the same snippet of code: It seems that bloggers copied that code from one another. This mistype is also long living: We located a page containing the error, `http://www.oocities.org/br/dicas.html/`, that is a mirror of a Brazilian Geocities site made in October 2009.

4.5 Countermeasures

In this section, we review two techniques that a web application can utilize to protect itself from malicious remotely-included scripts. Specifically, we examine the effectiveness of using a coarse-grained JavaScript sandboxing system and the option of creating local copies of remote JavaScript libraries.

4.5.1 Sandboxing remote scripts

Recognizing the danger of including a remote script, researchers have proposed a plethora of client-side and server-side systems that aim to limit the functionality of remotely-included JavaScript libraries (see Section 4.6). The majority of these countermeasures apply the principle of least privilege to remotely-included JavaScript code. More precisely, these systems attempt to limit the actions that can be performed by a remotely-included script to the bare minimum.

The least-privilege technique requires, for each remotely-included JavaScript file, a profile describing which functionality is needed when the script is executed. This profile can be generated either through manual code inspection or by first allowing the included script to execute and then recording all functions and properties of the Document Object Model (DOM) and Browser Object Model (BOM) that the script accessed. Depending on the sandboxing mechanism, these profiles can be either coarse-grained or fine-grained.

In a coarse-grained sandboxing system, the profile-writer instructs the sandbox to either forbid or give full access to any given resource, such as forbidding a script to use `eval`. Constrastingly, in a fine-grained sandboxing system, the profile-writer is able to instruct the sandbox to give access to only parts of resources to a remotely included script. For instance, using ConScript [103], a profile-writer can allow the dynamic creation of all types of elements except iframes, or allow the use of `eval` but only for the unpacking of JSON data. While this approach provides significantly more control over each script than a coarse-grained profile, it also requires more effort to describe correct and exact profiles. Moreover, each profile would need to be updated, every time that a remote script *legitimately* changes in a way that affects its current profile.

Static and dynamic analysis have been proposed as ways of automatically constructing profiles for sandboxing systems, however, they both have limitations in the coverage and correctness of the profiles that they can create. Static analysis cannot account for dynamically-loaded content, and dynamic analysis cannot account for code paths that were not followed in the training phase of the analysis. Moreover, even assuming a perfect code-coverage during training,

JS Action	# of Top scripts
Reading Cookies	41
`document.write()`	36
Writing Cookies	30
`eval()`	28
`XHR`	14
Accessing LocalStorage	3
Accessing SessionStorage	0
Geolocation	0

Table 4.7: JavaScript functionality used by the 100 most popularly included remote JavaScript files

it is non-trivial to automatically identify the particular use of each requested resource in order to transit from coarse-grained sandboxing to fine-grained.

Given this complex, error-prone and time-consuming nature of constructing fine-grained profiles, we wanted to assess whether coarse-grained profiles would sufficiently constrain popular scripts. To this end, we automatically generated profiles for the 100 most included JavaScript files, discovered through our crawl. If the privileges/resources required by legitimate scripts include everything that an attacker needs to launch an attack, then a coarse-grained sandboxing mechanism would not be an effective solution.

The actions performed by an included JavaScript file were discovered using the following setup: A proxy was placed in between a browser and the Internet. All traffic from the web browser was routed through the web proxy [64], which we modified to intercept HTTP traffic and inject instrumentation code into the passing HTML pages. This instrumentation code uses JavaScript's `setters` and `getters` to add wrappers to certain sensitive JavaScript functions and DOM/BOM properties, allowing us to monitor their use. The browser-provided on-demand stack-tracing functionality, allowed us to determine, at the time of execution of our wrappers, the chain of function calls that resulted in a specific access of a monitored resource. If a function, executed by a remote script, was part of this chain, then we safely deduce that the script was responsible for the activity, either by directly accessing our monitored resources or by assisting the access of other scripts.

For instance, suppose that a web page loads `a.js` and `b.js` as follows:

```
/* a.js */
function myalert(msg) {
    window.alert(msg);
}
```

```
/* b.js */
myalert("hello");
```

```
/* stack trace */
b.js:1:myalert(...)
a.js:2:window.alert(...)
```

In `a.js`, a function `myalert` is defined, which passes its arguments to the `window.alert()` function. Suppose `b.js` then calls `myalert()`. At the time this function is executed, the *wrapped* `window.alert()` function is executed. At this point, the stack trace contains both `a.js` and `b.js`, indicating that both are involved in the call to `window.alert()` (a potentially-sensitive function) and thus both can be held responsible. These accesses can be straightforwardly transformed into profiles, which can then be utilized by coarse-grained sandboxing systems.

Using the aforementioned setup, we visited web pages that included the top 100 most-included JavaScript files and monitored the access to sensitive JavaScript methods, DOM/BOM properties. The results of this experiment, presented in Table 4.7, indicate that the bulk of the most included JavaScript files read and write cookies, make calls to `document.write()`, and dynamically evaluate code from strings. Newer APIs on the other hand, like `localStorage`, `sessionStorage` and `Geolocation`, are hardly ever used, most likely due to their relatively recent implementation in modern web browsers.

The results show that, for a large part of the included scripts, it would be impossible for a coarse-grained sandboxing system to differentiate between benign and malicious scripts solely on their usage of cookie functionality. For instance, a remotely-included benign script that needs to access cookies to read and write identifiers for user-tracking can be substituted for a malicious script that leaks the including site's session identifiers. Both of these scripts access the same set of resources, yet the second one has the possibility of fully compromising the script-including site. It is also important to note that, due to the use of dynamic analysis and the fact that some code-paths of the executed scripts may not have been followed, our results are lower bounds of the scripts' access to resources, i.e., the tracked scripts may need access to more resources to fully execute.

Overall, our results highlight the fact that even in the presence of a coarse-grained sandboxing system that forbids unexpected accesses to JavaScript and browser resources, an attacker could still abuse the access already white-listed in the attacked script's profile. This means that regardless of their complexity, fine-grained profiles would be required in the majority of cases. We believe that this result motivates further research in fine-grained sandboxing and specifically in the automatic generation of correct script profiles.

4.5.2 Using local copies

Another way that web sites can avoid the risk of malicious script inclusions is by simply not including any remote scripts. To this end, a site needs to create local copies of remote JavaScript resources and then use these copies in their script inclusions. The creation of a local copy separates the security of the remote site from the script-including one, allowing the latter to be unaffected by a future compromise of the former. At the same time, however, this shifts the burden of updates to the developer of the script-including site who must verify and create a new local copy of the remote JavaScript library whenever a new version is made available.

To quantify the overhead of this manual procedure on the developer of a script-including web application, we decided to track the updates of the top 1,000 most-included scripts over the period of one week. This experiment was conducted four months after our large-scale crawling experiment, thus some URLs were no longer pointing to valid scripts. More precisely, from the top 1,000 scripts we were able to successfully download 803. We started by downloading this set three consecutive times within the same hour and comparing the three versions of each script. If a downloaded script was different all three times then we assume that the changes are not due to actual updates of the library, such as bug fixes or the addition of new functionality, but due to the embedding of constantly-changing data, such as random tokens, dates, and execution times. From this experiment, we found that 3.99% of our set of JavaScript scripts, seem to embed such data and thus appear to be constantly modified. For the rest of the experiment, we stopped tracking these scripts and focused on the ones that were identical all three times.

Over a period of one week, 10.21% of the monitored scripts were modified. From the modified scripts, 6.97% were modified once, 1.86% were modified twice, and 1.36% were modified three or more times. This shows that while some scripts undergo modifications more than once a week, 96.76% are modified at most once. We believe that the weekly manual inspection of a script's modified code is an acceptable tradeoff between increased maintenance time and increased

security of the script-including web application. At the same time, a developer who currently utilizes frequently-modified remote JavaScript libraries, might consider substituting these libraries for others of comparable functionality and less frequent modifications.

4.6 Related Work

Measurement Studies To the best of our knowledge, there has been no study of remote JavaScript inclusions and their implications that is of comparable breadth to our work. Yue and Wang conducted the first measurement study of insecure JavaScript practices on the web [223]. Using a set of 6,805 homepages of popular sites, they counted the sites that include remote JavaScript files, use the `eval` function, and add more information to the DOM of a page using `document.write`. Contrastingly, in our study, we crawled more than 3 million pages of the top 10,000 popular web sites, allowing us to capture five hundred times more inclusions and record behavior that is not necessarily present on a site's homepage. Moreover, instead of treating all remote inclusions as uniformly dangerous, we attempt to characterize the quality of their providers so that more trustworthy JavaScript providers can be utilized when a remote inclusion is unavoidable.

Richards et al. [154] and Ratanaworabhan et al. [146] study the dynamic behavior of popular JavaScript libraries and compare their findings with common usage assumptions of the JavaScript language and the functionality tested by common benchmarks. However, this is done without particular focus on the security features of the language. Richarts et al. [153] have also separately studied the use of `eval` in popular web sites.

Ocariza et al. [132] performed an empirical study of JavaScript errors in the top 100 Alexa sites. Seeking to quantify the reliability of JavaScript code in popular web applications, they recorded errors falling into four categories: "Permission Denied," "Null Exception," "Undefined Symbol" and "Syntax Error." Additionally, the authors showed that in some cases the errors were non-deterministic and depended on factors such as the speed of a user's interaction with the web application. The authors did not encounter any of the new types of vulnerabilities we described in Section 4.4, probably due to the limited size of their study.

Limiting available JavaScript functionality Based on the characterization of used functionality, included JavaScript files could be executed in a restricted environment that only offers the required subset of functionality. As we showed

in Section 4.5.1, a fine-grained sandboxing system is necessary because of the inability of coarse-grained sandboxes to differentiate between legitimate and malicious access to resources.

BrowserShield [148] is a server-side rewriting technique that replaces certain JavaScript functions to use safe equivalents. These safe equivalents are implemented in the “bshield” object that is introduced through the BrowserShield JavaScript libraries and injected into each page. BrowserShield makes use of a proxy to inject its code into a web page. Self-protecting JavaScript [140, 95] is a client-side wrapping technique that applies wrappers around JavaScript functions, without requiring any browser modifications. The wrapping code and policies are provided by the server and are executed first, ensuring a clean environment to start from.

ConScript [103] allows the enforcement of fine-grained security policies for JavaScript in the browser. The approach is similar to self-protecting JavaScript, except that ConScript modifies the browser to ensure that an attacker cannot abuse the browser-specific DOM implementation to find an unprotected access path. WebJail [192] is a client-side security architecture that enforces secure composition policies specified by a web-mashup integrator on third-party web-mashup components. Inspired by ConScript, WebJail modifies the Mozilla Firefox browser and JavaScript engine, to enforce these secure composition policies inside the browser. The new “sandbox” attribute of the iframe element in HTML5 [63] provides a way to limit JavaScript functionality, but it is very coarse-grained. It only supports limited restrictions, and as far as JavaScript APIs are concerned, it only supports to completely enable or disable JavaScript.

ADJail [184] is geared toward securely isolating ads from a hosting page for confidentiality and integrity purposes, while maintaining usability. The ad is loaded on a shadow page that contains only those elements of the hosting page to which the web developer wishes the ad to have access. Changes to the shadow page are replicated to the hosting page if those changes conform to the specified policy. Likewise, user actions on the hosting page are mimicked to the shadow page if allowed by the policy.

FlowFox [34] uses the related technique of secure multi-execution [36] to execute arbitrary included scripts with strong guarantees that these scripts can not break a specified confidentiality policy.

Content Security Policy (CSP) [179] is a mechanism that allows web application developers to specify from which locations their web application is allowed to load additional resources. Using CSP, a web application could be limited to only load JavaScript files from a specific set of third-party locations. In the case of typos in the URL, a CSP policy not containing that same typo will

prevent a JavaScript file from being loaded from that mistyped URL. Cases where a JavaScript-hosting site has been compromised and is serving malicious JavaScript however, will not be stopped by CSP.

AdSentry [39] is a confinement solution for JavaScript-based advertisement scripts. It consists of a shadow JavaScript engine which is used to execute untrusted JavaScript advertisements. Instead of having direct access to the DOM and sensitive functions, access from the shadow JavaScript engine is mediated by an access control policy enforcement subsystem.

4.7 Conclusion

Web sites that include JavaScript from remote sources in different administrative domains open themselves to attacks in which malicious JavaScript is sent to unsuspecting users, possibly with severe consequences. In this chapter, we extensively evaluated the JavaScript remote inclusion phenomenon, analyzing it from different points of view. We first determined how common it is to include remote JavaScript code among the most popular web sites on the Internet. We then provided an empirical evaluation of the quality-of-maintenance of these "code providers," according to a number of indicators. The results of our experiments show that indeed there is a considerable number of high-profile web sites that include JavaScript code from external sources that are not taking all the necessary security-related precautions and thus could be compromised by a determined attacker. As a by-product of our experiments, we identified several attacks that can be carried out by exploiting failures in the configuration and provision of JavaScript code inclusions. Our findings shed some light into the JavaScript code provider infrastructure and the risks associated with trusting external parties in implementing web applications.

Chapter 5

Web-based Device Fingerprinting

Preamble

This chapter presents a four-pronged analysis of web-based device fingerprinting where the entire fingerprinting ecosystem is evaluated. Our work sheds light into the current practices of commercial fingerprinting and highlights the difficulty of hiding one's browser identity. The contents of this chapter are replicated from the paper titled "Cookieless Monster: Exploring the Ecosystem of Web-based Device Fingerprinting" [125], which was published in the proceedings of the 34th IEEE Symposium of Security and Privacy (IEEE S&P), in 2013. This work was done with the collaboration of other authors from KU Leuven and the University of California, Santa Barbara. Nick Nikiforakis was the lead author of this paper.

5.1 Introduction

In 1994, Lou Montulli, while working for Netscape Communications, introduced the idea of cookies in the context of a web browser [185]. The cookie mechanism allows a web server to store a small amount of data on the computers of visiting users, which is then sent back to the web server upon subsequent requests. Using this mechanism, a website can build and maintain state over the otherwise stateless HTTP protocol. Cookies were quickly embraced by browser vendors

and web developers. Today, they are one of the core technologies on which complex, stateful web applications are built.

Shortly after the introduction of cookies, abuses of their stateful nature were observed. Web pages are usually comprised of many different resources, such as HTML, images, JavaScript, and CSS, which can be located both on the web server hosting the main page as well as other third-party web servers. With every request toward a third-party website, that website has the ability to set and read previously-set cookies on a user's browser. For instance, suppose that a user browses to *travel.com*, whose homepage includes a remote image from *tracking.com*. Therefore, as part of the process of rendering *travel.com*'s homepage, the user's browser will request the image from *tracking.com*. The web server of *tracking.com* sends the image along with an HTTP Set-Cookie header, setting a cookie on the user's machine, under the *tracking.com* domain. Later, when the user browses to other websites affiliated with *tracking.com*, e.g., *buy.com*, the tracking website receives its previously-set cookies, recognizes the user, and creates a profile of the user's browsing habits. These *third-party cookies*, due to the adverse effects on a user's privacy and their direct connection with online behavioral advertising, captured the attention of both the research community [87, 88, 158] and the popular media outlets [186] and, ever since, cause the public's discomfort [190, 191].

The user community responded to this privacy threat in multiple ways. A recent cookie-retention study by comScore [32] showed that approximately one in three users delete both first-party and third-party cookies within a month after their visit to a website. Multiple browser-extensions are available that reveal third-party tracking [56], as well as the "hidden" third-party affiliations between sites [31]. In addition, modern browsers now have native support for the rejection of all third-party cookies and some even enable it by default. Lastly, a browser's "Private Mode" is also available to assist users to visit a set of sites without leaving traces of their visit on their machine.

This general unavailability of cookies motivated advertisers and trackers to find new ways of linking users to their browsing histories. Mayer in 2009 [101] and Eckersley in 2010 [43] both showed that the features of a browser and its plugins can be fingerprinted and used to track users without the need of cookies. Today, there is a small number of commercial companies that use such methods to provide *device identification* through web-based fingerprinting. Following the classification of Mowery et al. [107], fingerprinting can be used either constructively or destructively. Constructively, a correctly identified device can be used to combat fraud, e.g., by detecting that a user who is trying to login to a site is likely an attacker who stole a user's credentials or cookies, rather than the legitimate user. Destructively, device identification through fingerprinting can be used to track users between sites, without their knowledge and without

a simple way of opting-out. Additionally, device identification can be used by attackers in order to deliver exploits, tailored for specific combinations of browsers, plugins and operating systems [84]. The line between the constructive and destructive use is, however, largely artificial, because the same technology is used in both cases.

Interestingly, companies were offering fingerprinting services as early as 2009, and experts were already voicing concerns over their impact on user privacy [105]. Even when fingerprinting companies honor the recently-proposed "Do Not Track" (DNT) header, the user is still fingerprinted for fraud detection, but the companies *promise* not to use the information for advertising purposes [21]. Note that since the fingerprinting scripts will execute regardless of the DNT value, the verification of this promise is much harder than verifying the effect of DNT on stateful tracking, where the effects are visible at the client-side, in a user's cookies [100].

In this chapter, we perform a four-pronged analysis of device identification through web-based fingerprinting. First, we analyze the fingerprinting code of three large, commercial companies. We focus on the differences of their code in comparison to Panopticlick [43], Eckersley's "open-source" implementation of browser fingerprinting. We identify the heavy use of Adobe Flash as a way of retrieving more sensitive information from a client, including the ability to detect HTTP proxies, and the existence of intrusive fingerprinting plugins that users may unknowingly host in their browsers. Second, we measure the adoption of fingerprinting on the Internet and show that, in many cases, sites of dubious nature fingerprint their users, for a variety of purposes. Third, we investigate special JavaScript-accessible browser objects, such as `navigator` and `screen`, and describe novel fingerprinting techniques that can accurately identify a browser even down to its minor version. These techniques involve the ordering of methods and properties, detection of vendor-specific methods, HTML/CSS functionality as well as minor but fingerprintable implementation choices. Lastly, we examine and test browser extensions that are available for users who wish to spoof the identity of their browser and show that, unfortunately *all* fail to completely hide the browser's true identity. This incomplete coverage not only voids the extensions but, ironically, also allows fingerprinting companies to detect the fact that user is attempting to hide, adding extra fingerprintable information.

Our main contributions are:

- We shed light into the current practices of device identification through web-based fingerprinting and propose a taxonomy of fingerprintable information.

- We measure the adoption of fingerprinting on the web.
- We introduce novel browser-fingerprinting techniques that can, in milliseconds, uncover a browser's family and version.
- We demonstrate how over 800,000 users, who are currently utilizing user-agent-spoofing extensions, are more fingerprintable than users who do not attempt to hide their browser's identity, and challenge the advice given by prior research on the use of such extensions as a way of increasing one's privacy [222].

5.2 Commercial Fingerprinting

While Eckersley showed the principle possibility of fingerprinting a user's browser in order to track users without the need of client-side stateful identifiers [43], we wanted to investigate popular, real-world implementations of fingerprinting and explore their workings. To this end, we analyzed the fingerprinting libraries of three large, commercial companies: BlueCava[1], Iovation[2] and ThreatMetrix.[3] Two of these companies were chosen due to them being mentioned in the web-tracking survey of Mayer and Mitchell [102], while the third one was chosen due to its high ranking on a popular search engine. Given the commercial nature of the companies, in order to analyze the fingerprinting scripts we first needed to discover websites that make use of them. We used Ghostery [56], a browser-extension which lists known third-party tracking libraries on websites, to obtain the list of domains which the three code providers use to serve their fingerprinting scripts. Subsequently, we crawled popular Internet websites, in search for code inclusions, originating from these fingerprinting-owned domains. Once these web sites were discovered, we isolated the fingerprinting code, extracted all individual features, and grouped similar features of each company together.

[1] `http://www.bluecava.com`
[2] `http://www.iovation.com`
[3] `http://www.threatmetrix.com`

Fingerprinting Category	Panopticlick	BlueCava	Iovation ReputationManager	ThreatMetrix
Browser customizations	Plugin enumeration$_{\text{(JS)}}$ Mime-type enumeration$_{\text{(JS)}}$ ActiveX + 8 CLSIDs$_{\text{(JS)}}$	Plugin enumeration$_{\text{(JS)}}$ ActiveX + 53 CLSIDs$_{\text{(JS)}}$ Google Gears Detection$_{\text{(JS)}}$		Plugin enumeration$_{\text{(JS)}}$ Mime-type enumeration$_{\text{(JS)}}$ ActiveX + 6 CLSIDs$_{\text{(JS)}}$ Flash Manufacturer$_{\text{(FLASH)}}$
Browser-level user configurations	Cookies enabled$_{\text{(HTTP)}}$ Timezone$_{\text{(JS)}}$ Flash enabled$_{\text{(JS)}}$	System/Browser/User Language$_{\text{(JS)}}$ Timezone$_{\text{(JS)}}$ Flash enabled$_{\text{(JS)}}$ Do-Not-Track User Choice$_{\text{(JS)}}$ MSIE Security Policy$_{\text{(JS)}}$	Browser Language$_{\text{(HTTP, JS)}}$ Timezone$_{\text{(JS)}}$ Flash enabled$_{\text{(JS)}}$ Date & time$_{\text{(JS)}}$ Proxy Detection$_{\text{(FLASH)}}$	Browser Language$_{\text{(FLASH)}}$ Timezone$_{\text{(JS, FLASH)}}$ Flash enabled$_{\text{(JS)}}$ Proxy Detection$_{\text{(FLASH)}}$
Browser family & version	User-agent$_{\text{(HTTP)}}$ ACCEPT-Header$_{\text{(HTTP)}}$ Partial S.Cookie test$_{\text{(JS)}}$	User-agent$_{\text{(JS)}}$ Math constants$_{\text{(JS)}}$ AJAX Implementation$_{\text{(JS)}}$	User-agent$_{\text{(HTTP, JS)}}$	User-agent$_{\text{(JS)}}$
Operating System & Applications	User-agent$_{\text{(HTTP)}}$ Font Detection$_{\text{(FLASH, JAVA)}}$	User-agent$_{\text{(JS)}}$ Font Detection$_{\text{(JS, FLASH)}}$ Windows Registry$_{\text{(SFP)}}$	User-agent$_{\text{(HTTP, JS)}}$ Windows Registry$_{\text{(SFP)}}$ MSIE Product key$_{\text{(SFP)}}$	User-agent$_{\text{(JS)}}$ Font Detection$_{\text{(FLASH)}}$ OS+Kernel version$_{\text{(FLASH)}}$
Hardware & Network	Screen Resolution$_{\text{(JS)}}$	Screen Resolution$_{\text{(JS)}}$ Driver Enumeration$_{\text{(SFP)}}$ IP Address$_{\text{(HTTP)}}$ TCP/IP Parameters$_{\text{(SFP)}}$	Screen Resolution$_{\text{(JS)}}$ Device Identifiers$_{\text{(SFP)}}$ TCP/IP Parameters$_{\text{(SFP)}}$	Screen Resolution$_{\text{(JS, FLASH)}}$

Table 5.1: Taxonomy of all features used by Panopticlick and the studied fingerprinting providers - shaded features are, in comparison to Panopticlick, either sufficiently extended, or acquired through a different method, or entirely new

In this section, we present the results of our analysis, in the form of a taxonomy of possible features that can be acquired through a fingerprinting library. This taxonomy covers all the features described in Panopticlick [43] as well as the features used by the three studied fingerprinting companies. Table 5.1 lists all our categories and discovered features, together with the method used to acquire each feature. The categories proposed in our taxonomy resulted by viewing a user's fingerprintable surface as belonging to a layered system, where the "application layer" is the browser and any fingerprintable in-browser information. At the top of this taxonomy, scripts seek to fingerprint and identify any browser customizations that the user has directly or indirectly performed. In lower levels, the scripts target user-specific information around the browser, the operating system and even the hardware and network of a user's machine. In the rest of this section, we focus on all the non-trivial techniques used by the studied fingerprinting providers that were not previously described in Eckersley's Panopticlick [43].

5.2.1 Fingerprinting through popular plugins

As one can see in Table 5.1, all companies use Flash, in addition to JavaScript, to fingerprint a user's environment. Adobe Flash is a proprietary browser plug-in that has enjoyed wide adoption among users, since it provided ways of delivering rich media content that could not traditionally be displayed using HTML. Despite the fact that Flash has been criticized for poor performance, lack of stability, and that newer technologies, like HTML5, can potentially deliver what used to be possible only through Flash, it is still available on the vast majority of desktops.

We were surprised to discover that although Flash reimplements certain APIs existing in the browser and accessible through JavaScript, its APIs do not always provide the same results compared to the browser-equivalent functions. For instance, for a Linux user running Firefox on a 64-bit machine, when querying a browser about the platform of execution, Firefox reports "Linux x86_64". Flash, on the other hand, provides the full kernel version, e.g., Linux 3.2.0-26-generic. This additional information is not only undesirable from a privacy perspective, but also from a security perspective, since a malicious web-server could launch an attack tailored not only to a browser and architecture but to a specific kernel. Another API call that behaves differently is the one that reports the user's screen resolution. In the Linux implementations of the Flash plugin (both Adobe's and Google's), when a user utilizes a dual-monitor setup, Flash reports as the width of a screen the sum of the two individual screens. This value, when combined with the browser's response (which lists the resolution of the monitor

where the browser-window is located), allows a fingerprinting service to detect the presence of multiple-monitor setups.

Somewhat surprisingly, none of the three studied fingerprinting companies utilized Java. One of them had some dead code that revealed that in the past it probably did make use of Java, however, the function was not called anymore and the applet was no longer present on the hard-coded location listed in the script. This is an interesting deviation from Panopticlick, which did use Java as an alternate way of obtaining system fonts. We consider it likely that the companies abandoned Java due to its low market penetration in browsers. This, in turn, is most likely caused by the fact that many have advised the removal of the Java plugin from a user's browser [29, 86] due to the plethora of serious Java vulnerabilities that were discovered and exploited over the last few years.

5.2.2 Vendor-specific fingerprinting

Another significant difference between the code we analyzed and Panopticlick is that, the fingerprinting companies were not trying to operate in the same way across all browsers. For instance, when recognizing a browser as Internet Explorer, they would extensively fingerprint Internet-Explorer-specific properties, such as `navigator.securityPolicy` and `navigator.systemLanguage`. At the same time, the code accounted for the browser's "short-comings," such as using a lengthy list of predefined CLSIDs for Browser-Helper-Objects (BHOs) due to Internet Explorer's unwillingness to enumerate its plugins.

5.2.3 Detection of fonts

The system's list of fonts can serve as part of a user's unique fingerprint [43]. While a browser does not directly provide that list, one can acquire it using either a browser plugin that willingly provides this information or using a side-channel that indirectly reveals the presence or absence of any given font.

Plugin-based detection

ActionScript, the scripting language of Flash, provides APIs that include methods for discovering the list of fonts installed on a running system. While this traditionally was meant to be used as a way of ensuring the correct appearance of text by the plugin, it can also be used to fingerprint the system. Two out of the three studied companies were utilizing Flash as a way of discovering which fonts were installed on a user's computer. Interestingly, only one of

Listing 1 Side-channel inference of the presence or absence of a font

```
function get_text_dimensions(font){

  h = document.getElementsByTagName("BODY")[0];
  d = document.createElement("DIV");
  s = document.createElement("SPAN");

  d.appendChild(s);
  d.style.fontFamily = font;
  s.style.fontFamily = font;
  s.style.fontSize = "72px";
  s.innerHTML = "font_detection";
  h.appendChild(d);

  textWidth = s.offsetWidth;
  textHeight = s.offsetHeight;
  h.removeChild(d);

  return [textWidth, textHeight];
}
```

the companies was preserving the order of the font-list, which points, most likely, to the fact that the other is unaware that the order of fonts is stable and machine-specific (and can thus be used as an extra fingerprinting feature).

Side-channel inference

The JavaScript code of one of the three fingerprinting companies included a fall-back method for font-detection, in the cases where the Flash plugin was unavailable. By analyzing that method, we discovered that they were using a technique, similar to the CSS history stealing technique [70], to identify the presence or absence of any given font - see Listing 1.

More precisely, the code first creates a `<div>` element. Inside this element, the code then creates a `<span>` element with a predetermined text string and size, using a provided font family. Using the `offsetWidth` and `offsetHeight` methods of HTML elements, the script discovers the layout width and height of the element. This code is first called with a “sans” parameter, the font typically used by browsers as a fall-back, when another requested font is unavailable on

Font Family	String	Width x Height
Sans	font_detection	519x84
Arial	font_detection	452x83
Calibri	font_detection	416x83

Figure 5.1: The same string, rendered with different fonts, and its effects on the string's width and height, as reported by the Google Chrome browser

a user's system. Once the height and text for "sans" are discovered, another script goes over a predefined list of fonts, calling the `get_text_dimensions` function for each one. For any given font, if the current width or height values are different from the ones obtained through the original "sans" measurement, this means that the font does exist and was used to render the predefined text. The text and its size are always kept constant, so that if its width or height change, this change will only be due to the different font. Figure 5.1 shows three renderings of the same text, with the same font-size but different font faces in Google Chrome. In order to capitalize as much as possible on small differences between fonts, the font-size is always large, so that even the smallest of details in each individual letter will add up to measurable total difference in the text's height and width. If the height and width are identical to the original measurement, this means that the requested font did not exist on the current system and thus, the browser has selected the sans fall-back font. All of the above process, happens in an invisible iframe created and controlled by the fingerprinting script and thus completely hidden from the user.

Using this method, a fingerprinting script can rapidly discover, even for a long list of fonts, those that are present on the operating system. The downside of this approach is that less popular fonts may not be detected, and that the font-order is no longer a fingerprintable feature.

5.2.4 Detection of HTTP Proxies

One of the features that are the hardest to spoof for a client is its IP address. Given the nature of the TCP protocol, a host cannot pretend to be listening at an IP address from which it cannot reliably send and receive packets. Thus, to hide a user's IP address, another networked machine (a proxy) is typically employed that relays packets between the user who wishes to remain hidden and a third-party. In the context of browsers, the most common type of proxies are HTTP proxies, through which users configure their browsers to send all

requests. In addition to manual configuration, browser plugins are also available that allow for a more controlled use of remote proxies, such as the automatic routing of different requests to different proxies based on pattern matching of each request,[4] or the cycling of proxies from a proxy list at user-defined intervals.[5]

From the point of view of device identification through fingerprinting, a specific IP address is an important feature. Assuming the use of fingerprinting for the detection of fraudulent activities, the distinction between a user who is situated in a specific country and one that *pretends* to be situated in that country, is crucial. Thus, it is in the interest of the fingerprint provider to detect a user's real IP address or, at least, discover that the user is utilizing a proxy server.

When analyzing the ActionScript code embedded in the SWF files of two of the three fingerprinting companies, we found evidence that the code was circumventing the user-set proxies at the level of the browser, i.e., the loaded Flash application was contacting a remote host directly, disregarding any browser-set HTTP proxies. We verified this behavior by employing both an HTTP proxy and a packet-capturing application, and noticing that certain requests were captured by the latter but were never received by the former. In the code of both of the fingerprinting companies, certain long alphanumerical tokens were exchanged between JavaScript and Flash and then used in their communication to the server. While we do not have access to the server-side code of the fingerprinting providers, we assume that the identifiers are used to correlate two possibly different IP addresses. In essence, as shown in Figure 5.2, if a JavaScript-originating request contains the same token as a Flash-originating request from a different source IP address, the server can be certain that the user is utilizing an HTTP proxy.

Flash's ability to circumvent HTTP proxies is a somewhat known issue among privacy-conscious users that has lead to the disabling of Flash in anonymity-providing applications, like TorButton [189]. Our analysis shows that it is actively exploited to identify and bypass web proxies.

5.2.5 System-fingerprinting plugins

Previous research on fingerprinting a user's browser focused on the use of popular browser plugins, such as Flash and Java, and utilized as much of their API surface as possible to obtain user-specific data [101, 43]. However, while analyzing the plugin-detection code of the studied fingerprinting providers, we

[4] FoxyProxy - `http://getfoxyproxy.org/`
[5] ProxySwitcher - `http://www.proxyswitcher.com/`

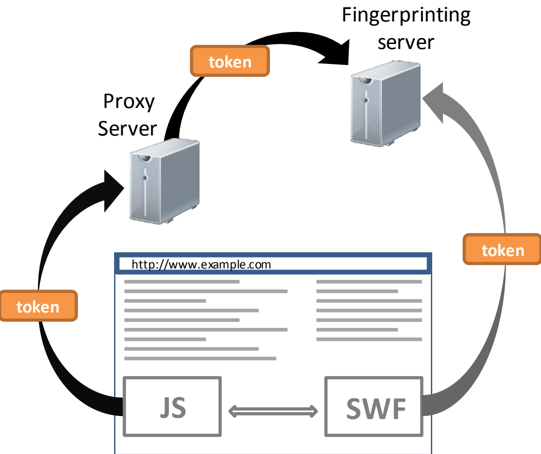

Figure 5.2: Fingerprinting libraries take advantage of Flash's ability to ignore browser-defined HTTP proxies to detect the real IP address of a user

noticed that two out of the three were searching a user's browser for the presence of a special plugin, which, if detected, would be loaded and then invoked. We were able to identify that the plugins were essentially native fingerprinting libraries, which are distributed as CAB files for Internet Explorer and eventually load as DLLs inside the browser. These plugins can reach a user's system, either by a user accepting their installation through an ActiveX dialogue, or bundled with applications that users download on their machines. DLLs are triggered by JavaScript through ActiveX, but they run natively on the user's machine, and thus can gather as much information as the Internet Explorer process.

We downloaded both plugins, wrapped each DLL into an executable that simply hands-off control to the main routine in the DLL and submitted both executables to Anubis [9], a dynamic malware analysis platform that executes submitted binaries in a controlled environment. We focused on the Windows registry values that were read by the plugin, since the registry is a rich environment for fingerprinting. The submitted fingerprinting DLLs were reading a plethora of system-specific values, such as the hard disk's identifier, TCP/IP parameters, the computer's name, Internet Explorer's product identifier, the installation date of Windows, the Windows Digital Product Id and the installed system drivers – entries marked with SFP in Table 5.1.

All of these values combined provide a much stronger fingerprint than what JavaScript or Flash could ever construct. It is also worth mentioning that one of the two plugins was misleadingly identifying itself as "ReputationShield" when

asking the user whether she wants to accept its installation. Moreover, none of 44 antivirus engines of VirusTotal [193] identified the two DLLs as malicious, even though they clearly belong to the *spyware* category. Using identifiers found within one DLL, we were also able to locate a Patent Application for Iovation's fingerprinting plugin that provides further information on the fingerprinting process and the gathered data [141].

5.2.6 Fingerprint Delivery Mechanism

In the fingerprinting experiments of Mayer [101] and Eckersley [43], there was a 1-to-1 relationship between the page conducting the fingerprinting and the backend storing the results. For commercial fingerprinting, however, there is a N-to-1 relationship, since each company provides fingerprinting services to many websites (through the inclusion of third-party scripts) and needs to obtain user fingerprints from each of these sites. Thus, the way that the fingerprint and the information about it are delivered is inherently different from the two aforementioned experiments.

Through our code analysis, we found two different scenarios of fingerprinting. In the first scenario, the first-party site was not involved in the fingerprinting process. The fingerprinting code was delivered by an advertising syndicator, and the resulting fingerprint was sent back to the fingerprinting company. This was most likely done to combat click-fraud, and it is unclear whether the first-party site is even aware of the fact that its users are being fingerprinted.

In the second scenario, where the first-party website is the one requesting the fingerprint, we saw that two out of the three companies were adding the final fingerprint of the user into the DOM of the hosting page. For instance, `www.imvu.com` is using BlueCava for device fingerprinting by including remote scripts hosted on BlueCava's servers. When BlueCava's scripts combine all features into a single fingerprint, the fingerprint is DES-encrypted (DES keys generated on the fly and then encrypted with a public key), concatenated with the encrypted keys and finally converted to Base64 encoding. The resulting string is added into the DOM of `www.imvu.com`; more precisely, as a new hidden input element in IMVU's login form. In this way, when the user submits her username and password, the fingerprint is also sent to IMVU's web servers. Note, however, that IMVU cannot decrypt the fingerprint and must thus submit it back to BlueCava, which will then reply with a "trustworthiness" score and other device information. This architecture allows BlueCava to hide the implementation details from its clients and to correlate user profiles across its entire client-base. Iovation's fingerprinting scripts operate in a similar manner.

Constrastingly, ThreatMetrix delivers information about users in a different way. The including site, i.e., a customer of ThreatMetrix, creates a session identifier that it places into a `<div>` element with a predefined identifier. ThreatMetrix's scripts, upon loading, read this session identifier and append it to all requests towards the ThreatMetrix servers. This means that the including site never gets access to a user's fingerprint, but only information about the user by querying ThreatMetrix for specific session identifiers.

5.2.7 Analysis Limitations

In the previous sections we analyzed the workings of the fingerprinting libraries of three popular commercial companies. The analysis was a mostly manual, time-consuming process, where each piece of code was gradually deobfuscated until the purpose of all functions was clear. Given the time required to fully reverse-engineer each library, we had to limit ourselves to analyze the script of each fingerprinting company as it was seen through two different sites (that is, two different clients of each company). However, we cannot exclude the possibility of additional scripts that are present on the companies' web servers that would perform more operations than the ones we encountered.

5.3 Adoption of fingerprinting

In Section 5.2, we analyzed the workings of three commercial fingerprinting companies and focused on the differences of their implementations when compared to Panopticlick [43]. In this section, we study the fingerprinting ecosystem, from the point of view of websites that leverage fingerprinting.

5.3.1 Adoption on the popular web

To quantify the use of web-based fingerprinting on popular websites, we crawled up to 20 pages for each of the Alexa top 10,000 sites, searching for script inclusions and iframes originating from the domains that the three studied companies utilize to serve their fingerprinting code. To categorize the discovered domains, we made use of the publicly-available domain categorization service of TrendMicro,[6] a popular anti-virus vendor.

Through this process, we discovered 40 sites (0.4% of the Alexa top 10,000) utilizing fingerprinting code from the three commercial providers. The most

[6] TrendMicro - `http://global.sitesafety.trendmicro.com/`

popular site making use of fingerprinting is *skype.com*, while the two most popular categories of sites are: "Pornography" (15%) and "Personals/Dating" (12.5%). For pornographic sites, a reasonable explanation is that fingerprinting is used to detect shared or stolen credentials of paying members, while for dating sites to ensure that attackers do not create multiple profiles for social-engineering purposes. Our findings show that fingerprinting is already part of some of the most popular sites of the Internet, and thus the hundreds of thousands of their visitors are fingerprinted on a daily basis.

Note that the aforementioned adoption numbers are lower bounds since our results do not include pages of the 10,000 sites that were not crawled, either because they were behind a registration wall, or because they were not in the set of 20 URLs for each crawled website. Moreover, some popular sites may be using their own fingerprinting algorithms for performing device identification and not rely on the three studied fingerprinting companies.

5.3.2 Adoption by other sites

To discover less popular sites making use of fingerprinting, we used a list of 3,804 domains of sites that, when analyzed by Wepawet [33], requested the previously identified fingerprinting scripts.

Each domain was submitted to TrendMicro's and McAfee's categorization services[7] which provided as output the domain's category and "safety" score. We used two categorizing services in an effort to reduce, as much as possible, the number of "untested" results, i.e., the number of websites not analyzed and not categorized. By examining the results, we extracted as many popular categories as possible and created aliases for names that were referring to the same category, such as "News / Media" versus "General News" and "Disease Vector" versus "Malicious Site". If a domain was characterized as "dangerous" by one, and "not dangerous" by the other, we accepted the categorization of the latter, so as to give the benefit of the doubt to legitimate websites that could have been compromised, when the former service categorized it.

Given the use of two domain-categorization services, a small number of domains (7.9%) was assigned conflicting categories, such as "Dating" versus "Adult/Mature" and "Business/Economy" versus "Software/Hardware." For these domains, we accepted the characterization of McAfee, which we observed to be more precise than TrendMicro's for less popular domains. Excluding 40.8% of domains which were reported as "untested" by both services, the results of this categorization are shown in Figure 5.3.

[7] McAfee - `http://mcafee.com/threat-intelligence/domain/`

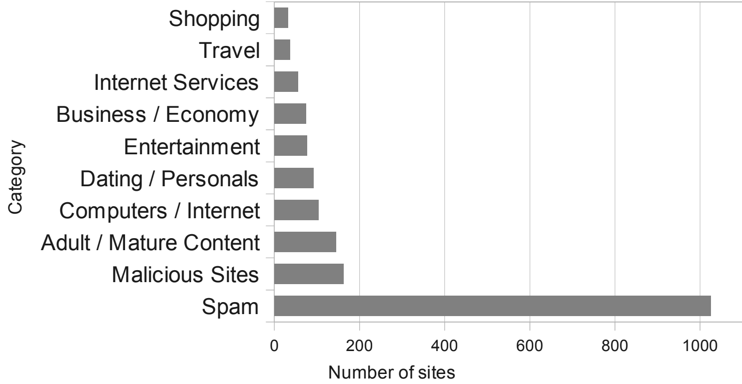

Figure 5.3: The top 10 categories of websites utilizing fingerprinting

First, one can observe that eight out of the ten categories, include sites which operate with user subscriptions, many of which contain personal and possibly financial information. These sites are usually interested in identifying fraudulent activities and the hijacking of user accounts. The Adult/Mature category seems to make the most use of fingerprinting as was the case with the Alexa top 10,000 sites.

The top two categories are also the ones that were the least expected. 163 websites were identified as malicious, such as using exploits for vulnerable browsers, conducting phishing attacks or extracting private data from users, whereas 1,063 sites were categorized as "Spam" by the two categorizing engines. By visiting some sites belonging to these categories, we noticed that many of them are parked webpages, i.e., they do not hold any content except advertising the availability of the domain name, and thus do not currently include fingerprinting code. We were however able to locate many "quiz/survey" sites that are, at the time of this writing, including fingerprinting code from one of the three studied companies. Visitors of these sites are greeted with a "Congratulations" message, which informs them that they have won and asks them to proceed to receive their prize. At some later step, these sites extract a user's personal details and try to subscribe the user to expensive mobile services.

While our data-set is inherently skewed towards "maliciousness" due to its source, it is important to point out that all of these sites were found to include, at some point in time, fingerprinting code provided by the three studied providers. This observation, coupled with the fact that for all three companies, an interested client must set an appointment with a sales representative in order to acquire fingerprinting services, point to the possibility of fingerprinting companies

working together with sites of dubious nature, possibly for the expansion of their fingerprint databases and the acquisition of more user data.

5.4 Fingerprinting the behavior of special objects

In Section 5.2, we studied how commercial companies perform their fingerprinting and created a taxonomy of fingerprintable information accessible through a user's browser. In Table 5.1, one can notice that, while fingerprinting companies go to great lengths to discover information about a browser's plugins and the machine hosting the browser, they mostly rely on the browser to willingly reveal its true identity (as revealed through the `navigator.userAgent` property and the User-Agent HTTP header). A browser's user-agent is an important part of a system's fingerprint [222], and thus it may seem reasonable to assume that if users modify these default values, they will increase their privacy by hiding more effectively from these companies.

In this section, however, we demonstrate how fragile the browser ecosystem is against fingerprinting. Fundamental design choices and differences between browser types are used in an effort to show how difficult it can be to limit the exposure of a browser to fingerprinting. Even different versions of the same browser can have differences in the scripting environment that identify the browser's real family, version, and, occasionally, even the operating system. In the rest of this section we describe several novel browser-identifying techniques that: a) can complement current fingerprinting, and b) are difficult to eliminate given the current architecture of web browsers.

5.4.1 Experimental Fingerprinting Setup

Our novel fingerprinting techniques focus on the special, browser-populated JavaScript objects; more precisely, the `navigator` and `screen` objects. Contrary to objects created and queried by a page's JavaScript code, these objects contain vendor- and environment-specific methods and properties, and are thus the best candidates for uncovering vendor-specific behaviors.

To identify differences between browser-vendors and to explore whether these differences are consistent among installations of the same browser on multiple systems, we constructed a fingerprinting script that performed a series of "everyday" operations on these two special objects (such as adding a new property to an object, or modifying an existing one) and reported the results to a server. In this and the following section, we describe the operations of

our fingerprinting script and our results. Our constructed page included a JavaScript program that performed the following operations:

1. Enumerated the `navigator` and `screen` object, i.e., request the listing of all properties of the aforementioned objects.
2. Enumerated the `navigator` object again, to ensure that the order of enumeration does not change.
3. Created a custom object, populated it, and enumerated it. A custom, JavaScript-created object, allows us to compare the behavior of browser-populated objects (such as `navigator`) with the behavior of "classic" JavaScript objects.
4. Attempted to delete a property of the `navigator` object, the `screen` object, and the custom object.
5. Add the possibly-deleted properties back to their objects.
6. Attempted to modify an existing property of the `navigator` and `screen` objects.
7. If `Object.defineProperty` is implemented in the current browser, utilize it to make an existing property in the `navigator`, `screen`, and custom object non-enumerable.
8. Attempt to delete the `navigator` and `screen` objects.
9. Attempt to assign new custom objects to the `navigator` and `screen` variable names.

At each step, the objects involved were re-enumerated, and the resulting data was Base64-encoded and sent to our server for later processing. Thus, at the server side, we could detect whether a property was deleted or modified, by comparing the results of the original enumeration with the current one. The enumeration of each object was conducted through code that made use of the *prop in obj* construct, to avoid forcing a specific order of enumeration of the objects, allowing the engine to list object properties in the way of its choosing.

5.4.2 Results

By sharing the link to our fingerprinting site with friends and colleagues, we were able, within a week, to gather data from 68 different browsers installations,

of popular browsers on all modern operating systems. While our data is small in comparison to previous studies [101, 43], we are not using it to draw conclusions that have statistical relevance but rather, as explained in the following sections, to find deviations between browsers and to establish the consistency of these deviations. We were able to identify the following novel ways of distinguishing between browsers:

Order of enumeration Through the analysis of the output from the first three steps of our fingerprinting algorithm (Sec. 5.4.1), we discovered that the order of property-enumeration of special browser objects, like the `navigator` and `screen` objects, is consistently different between browser families, versions of each browser, and, in some cases, among deployments of the same version on different operating systems. While in the rest of this section we focus to the `navigator` object, the same principles apply to the `screen` object.

Our analysis was conducted in the following manner. After grouping the `navigator` objects and their enumerated properties based on browser families, we located the `navigator` object with the least number of properties. This version was consistently belonging to the oldest version of a browser, since newer versions add new properties which correspond to new browser features, such as the `navigator.doNotTrack` property in the newer versions of Mozilla Firefox. The order of the properties of this object, became our baseline to which we compared the `navigator` objects of all subsequent versions of the same browser family. To account for ordering changes due to the introduction of new properties in the `navigator` object, we simply excluded all properties that were not part of our original baseline object, without however changing the relative order of the rest of the properties. For instance, assume an ordered set of features B, where $B_0 = \{a, b, c, d\}$ and $B_1 = \{a, b, \underline{e}, c, d, \underline{f}\}$. B_1 has two new elements in comparison with B_0, namely `e` and `f` which, however, can be removed from the set without disrupting the relative order of the rest. For every browser version within the same browser-family, we compared the `navigator` object to the baseline, by first recording and removing new features and then noting whether the order of the remaining features was different from the order of the baseline.

The results of this procedure are summarized in Table 5.2. For each browser family, we compare the ordering of the `navigator` object among up to five different versions. The most current version is denoted as V_c. The first observation is that in almost 20 versions of browsers, no two were ever sharing the same order of properties in the `navigator` object. This feature by itself, is sufficient to categorize a browser to its correct family, regardless of any

Browser	**V_{c-4}**	**V_{c-3}**	**V_{c-2}**	**V_{c-1}**	**V_c**
Mozilla Firefox	W	W+1	W+4	W+5	W+7
Microsoft IE	-	-	X	X	X+1
Opera	Y	Y+1	Y+1	Y+3	Y+5
Google Chrome	Z	Z	Z'+1	Z''+1	Z'''+1

Table 5.2: Differences in the order of `navigator` objects between versions of the same browser

property-spoofing that the browser may be employing. Second, all browsers except Chrome maintain the ordering of `navigator` elements between versions. Even when new properties were introduced, these do not alter the relative order of all other properties. For instance, even though the newest version of Mozilla Firefox (V_c) has 7 extra features when compared to the oldest version (V_{c-4}), if we ignore these features then the ordering is the same with the original ordering (W).

Google Chrome was the only browser that did not exhibit this behavior. By analyzing our dataset, we discovered that Chrome not only changed the order between subsequent versions of the browser, but also between deployments of the same browser on different operating systems. For instance, Google Chrome v.20.0.1132.57 installed on Mac OSX has a different order of elements than the same version installed on a Linux operating system. In Table 5.2, we compare the order of properties of the `navigator` object when the underlying OS is Windows XP. While this changing order may initially appear to be less-problematic than the stable order of other browsers, in reality, the different orderings can be leveraged to detect a specific version of Google Chrome, and, in addition, the operating system on which the browser is running.

Overall, we discovered that the property ordering of special objects, such as the `navigator` object, is consistent among runs of the same browser and runs of the same version of browsers on different operating systems. Contrastingly, the order of properties of a custom script-created object (Step 3 in Section 5.4.1) was identical among all the studied browsers even though, according to the ECMAScript specification, objects are *unordered* collections of properties [44] and thus the exact ordering can be implementation-specific. More precisely, the property ordering of the custom objects was always the same with the order of property creation.

In general, the browser-specific, distinct property ordering of special objects can be directly used to create models of browsers and, thus, unmask the real identity of a browser. Our findings are in par with the “order-matters” observation

Browser	Unique methods & properties
Mozilla Firefox	screen.mozBrightness screen.mozEnabled navigator.mozSms + 10
Google Chrome	navigator.webkitStartActivity navigator.getStorageUpdates
Opera	navigator.browserLanguage navigator.getUserMedia
Microsoft IE	screen.logicalXDPI screen.fontSmoothingEnabled navigator.appMinorVersion +11

Table 5.3: Unique methods and properties of the `navigator` and `screen` objects of the four major browser-families

made by previous research: Mayer discovered that the list of plugins as reported by browsers was ordered based on the installation time of each individual plugin [101]. Eckersley noticed that the list of fonts, as reported by Adobe Flash and Sun's Java VM, remained stable across visits of the same user [43].

Unique features During the first browser wars in the mid-90s, browser vendors were constantly adding new features to their products, with the hope that developers would start using them. As a result, users would have to use a specific browser, effectively creating a browser lock-in [225]. The features ranged from new HTML tags to embedded scripting languages and third-party plugins. Signs of this "browser battle" are still visible in the contents of the user-agent string of modern browsers [7].

Today, even though the HTML standard is governed by the W3C committee and JavaScript by Ecma International, browser vendors still add new features that do not belong to any specific standard. While these features can be leveraged by web developers to provide users with a richer experience, they can also be used to differentiate a browser from another. Using the data gathered by our fingerprinting script, we isolated features that were available in only one family of browsers, but not in any other. These unique features are summarized in Table 5.3. All browser families had at least two such features that were not shared by any other browser. In many cases, the names of the new features were starting with a vendor-specific prefix, such as `screen.mozBrightness` for

Mozilla Firefox and `navigator.msDoNotTrack` for Microsoft Internet Explorer. This is because browser-vendors are typically allowed to use prefixes for features not belonging to a standard or not yet standardized [196]. In the context of fingerprinting, a script can query for the presence or absence of these unique features (e.g., typeof screen.mozBrightness != "undefined") to be certain of the identity of any given browser.

An interesting sidenote is that these unique features can be used to expose the real version of Mozilla Firefox browser, even when the user is using the Torbutton extension. Torbutton replaces the `navigator` and `screen` objects with its own versions, spoofing the values of certain properties, so as to protect the privacy of the user [42]. We installed Torbutton on Mozilla Firefox version 14 and, by enumerating the `navigator` object, we observed that, among others, the Torbutton had replaced the `navigator.userAgent` property with the equivalent of Mozilla Firefox version 10, and it was claiming that our platform was Windows instead of Linux. At the same time, however, special Firefox-specific properties that Mozilla introduced in versions 11 to 14 of Firefox (such as `navigator.mozBattery` and `navigator.mozSms`) were still available in the `navigator` object. These discrepancies, combined with other weaknesses found in less thorough user-agent-spoofing extensions (see Section 5.5), can uncover not only that the user is trying to hide, but also that she is using Torbutton to do so.

Mutability of special objects In the two previous sections, we discussed the ability to exploit the enumeration-order and unique features of browsers for fingerprinting. In this section, we investigate whether each browser treats the `navigator` and `screen` objects like regular JavaScript objects. More precisely, we investigate whether these objects are mutable, i.e., whether a script can delete a specific property from them, replace a property with a new one, or delete the whole object. By comparing the outputs of steps four to nine from our fingerprinting algorithm, we made the following observations.

Among the four browser families, only Google Chrome allows a script to delete a property from the `navigator` object. In all other cases, while the "delete" call returns successfully and no exceptions are thrown, the properties remain present in the special object. When our script attempted to modify the value of a property of `navigator`, Google Chrome and Opera allowed it, while Mozilla Firefox and Internet Explorer ignored the request. In the same way, these two families were the only ones allowing a script to reassign `navigator` and `screen` to new objects. Interestingly, no browser allowed the script to simply delete the `navigator` or `screen` object. Finally, Mozilla Firefox behaved in a unique way when requested to make a certain property of the `navigator`

object non-enumerable. Specifically, instead of just hiding the property, Firefox behaved as if it had actually deleted it, i.e., it was no longer accessible even when requested by name.

Evolution of functionality Recently, we have seen a tremendous innovation in Web technologies. The competition is fierce in the browsers' scene, and vendors are trying hard to adopt new technologies and provide a better platform for web applications. Based on that observation, in this section, we examine if we can determine a browser's version based on the new functionality that it introduces. We chose Google Chrome as our testing browser and created a library in JavaScript that tests if specific functionality is implemented by the browser. The features that we selected to capture different functionality were inspired by web design compatibility tests (where web developers verify if their web application is compatible with a specific browser). In total, we chose 187 features to test in 202 different versions of Google Chrome, spanning from version *1.0.154.59* up to *22.0.1229.8*, which we downloaded from *oldapps.com* and which covered all 22 major versions of Chrome. We found that not all of the 187 features were useful; only 109 actually changed during Google Chrome's evolution. These browser versions covered not only releases from the stable channel of Google Chrome, but also from Beta and Dev channels. We refer to a major version as the first number of Google Chrome's versioning system, and to minor version as the full number of the version. We used a virtual machine with Windows XP to setup all browser versions, and used all versions to visit our functionality-fingerprinting page.

In total, we found 71 sets of features that can be used to identify a specific version of Google Chrome. Each feature set could identify versions that range from a single Google Chrome version up to 14 different versions. The 14 Chrome versions that were sharing the same feature set were all part of the *12.0.742.** releases. Among all 71 sets, there were only four cases where the same feature set was identifying more than a single major version of the browser. In all of these cases, the features overlapped with the first Dev release of the next major version, while subsequent releases from that point on had different features implemented. In Figure 5.4, we show how many minor versions of Chrome we examined per major version and how many distinct feature sets we found for each major version. The results show that we can not only identify the major version, but in most cases, we have several different feature sets on the same major version. This makes the identification of the exact browser version even more fine-grained.

In Figure 5.5, we show how one can distinguish all Google Chrome's major

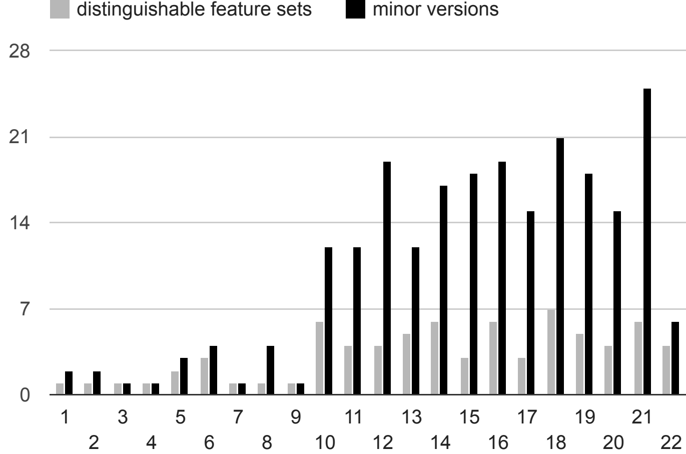

Figure 5.4: A comparison between how many distinguishable feature sets and minor Google Chrome versions we have per Google Chrome's major versions.

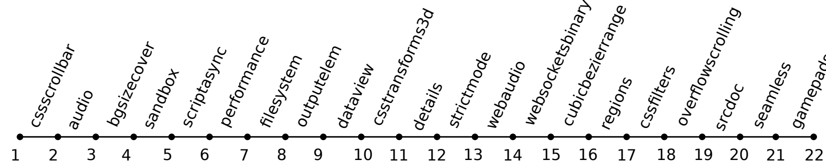

Figure 5.5: Feature-based fingerprinting to distinguish between Google Chrome major versions

versions by checking for specific features. Every pair of major versions is separated by a feature that was introduced into the newer version and did not exist in the previous one. Thus, if anyone wants to distinguish between two consecutive versions, a check of a single feature is sufficient to do so. Notice that our results indicate that we can perform even more fine-grained version detection than the major version of Google Chrome (we had 71 distinct sets of enabled features compared to 22 versions of Chrome), but for simplicity we examined only the major version feature changes in detail.

Miscellaneous In this section, we list additional browser-specific behaviors that were uncovered through our experiment but that do not fall in the previous categories.

Our enumeration of object-properties indirectly uses the method `toString()`

for the examined objects. By comparing the formatted output of some specific properties and methods, we noticed that different browsers treated them in slightly different ways. For instance, when calling `toString()` on the natively implemented `navigator.javaEnabled` method, browsers simply state that it is a "native function." Although all the examined browser families print "function javaEnabled() { [native code] }," Firefox uses newline characters after the opening curly-bracket and before the closing one. Interestingly, Internet Explorer does not list the `navigator.javaEnabled` when requested to enumerate the `navigator` object, but still provides the "native function" print-out when asked specifically about the `javaEnabled` method. In the same spirit, when our scripts invoked the `toString()` method on the `navigator.plugins` object, Google Chrome reported "[object DOMPluginArray]," Internet Explorer reported "[object]," while both Mozilla Firefox and Opera reported "[object PluginArray]."

Lastly, while trying out our fingerprinting page with various browsers, we discovered that Internet Explorer lacks native support for Base64 encoding and decoding (`atob` and `btoa`, respectively) which our script used to encode data before sending them to the server.

5.4.3 Summary

Overall, one can see how various implementation choices, either major ones, such as the traversal algorithms for JavaScript objects and the development of new features, or minor ones, such as the presence or absence of a newline character, can reveal the true nature of a browser and its JavaScript engine.

5.5 Analysis of User-Agent-Spoofing Extensions

With the advent of browser add-ons, many developers have created extensions that can increase the security of users (e.g., extensions showing HTML forms with non-secure destinations) or their privacy (e.g., blocking known ads and web-tracking scripts).

In the context of this chapter, we were interested in studying the completeness and robustness of extensions that attempt to hide the true nature of a browser from an inspecting website. As shown in Table 5.1, while the studied companies do attempt to fingerprint a user's browser customizations, they currently focus only on browser-plugins and do not attempt to discover any installed browser-extensions. Given however the sustained popularity of browser-extensions [166],

Extension	#Installations	User Rating
Mozilla Firefox		
UserAgent Switcher	604,349	4/5
UserAgent RG	23,245	4/5
UAControl	11,044	4/5
UserAgentUpdater	5,648	3/5
Masking Agent	2,262	4/5
User Agent Quick Switch	2,157	5/5
randomUserAgent	1,657	4/5
Override User Agent	1,138	3/5
Google Chrome		
User-Agent Switcher for Chrome	123,133	4/5
User-Agent Switcher	21,108	3.5/5
Ultimate User Agent Switcher, URL sniffer	28,623	4/5

Table 5.4: List of user-agent-spoofing browser extensions

we consider it likely that fingerprinting extensions will be the logical next step. Note that, unlike browser plugins, extensions are not enumerable through JavaScript and, thus, can only be detected through their side-effects. For instance, some sites currently detect the use of Adblock Plus [1] by searching for the absence of specific iframes and DOM elements that are normally created by advertising scripts.

Since a browser exposes its identity through the user-agent field (available both as an HTTP header and as a property of the JavaScript-accessible `navigator` object), we focused on extensions that advertised themselves as capable of spoofing a browser's user agent. These extensions usually serve two purposes. First, they allow users to surf to websites that impose strict browser requirements onto their visitors, without fulfilling these requirements. For instance, some sites are developed and tested using one specific browser and, due to the importance of the content loading correctly, refuse to load on other browsers. Using a user-agent-spoofing extension, a user can visit such a site, by pretending to use one of the white-listed browsers.

Another reason for using these extensions is to protect the privacy of a user. Eckeresly, while gathering data for the Panopticlick project, discovered that there were users whose browsers were reporting impossible configurations, for instance, a device was pretending to be an iPhone, but at the same time had Adobe

	Google Chrome	**Mozilla Firefox**	**MSIE**	**Opera**
navigator.product	Gecko	Gecko	N/A	N/A
navigator.appCodeName	Mozilla	Mozilla	Mozilla	Mozilla
navigator.appName	Netscape	Netscape	Microsoft Internet Explorer	Opera
navigator.platform	Linux i686	Linux x86_64	Win32	Linux
navigator.vendor	Google Inc.	(empty string)	N/A	N/A

Table 5.5: Standard properties of the navigator object and their values across different browser families

Flash support. In that case, these were users who were obviously trying to get a non-unique browser fingerprint by Panopticlick. Since Eckersley's study showed the viability of using common browser features as parts of a unique fingerprint, it is reasonable to expect that legitimate users utilize such extensions to reduce the trackability of their online activities, even if the extensions' authors never anticipated such a use. Recently, Trusteer discovered in an "underground" forum a spoofing-guide that provided step-by-step instructions for cybercriminals who wished to fool fraud-detection mechanisms that used device-fingerprinting [82]. Among other advice, the reader was instructed to download an extension that changes the User-Agent of their browser to make their sessions appear as if they were originating by different computers with different browsers and operating systems.

Table 5.4 shows the Mozilla Firefox and Google Chrome extensions that we downloaded and tested, together with their user base (measured in July 2012) and the rating that their users had provided. The extensions were discovered by visiting each market, searching for "user-agent" and then downloading all the relevant extensions with a sufficiently large user base and an above-average rating. A high rating is important because it indicates the user's satisfaction in the extension fulfilling its purpose. Our testing consisted of listing the `navigator` and `screen` objects through JavaScript and inspecting the HTTP headers sent with browser requests, while the extensions were actively spoofing the identity of the browser. As in Section 5.4, we chose to focus on these two objects since they are the ones that are the most vendor-specific as well as the most probed by the fingerprinting libraries. Through our analysis, we discovered that, unfortunately, in *all* cases, the extensions were inadequately hiding the real identity of the browser, which could still be straightforwardly exposed through

JavaScript. Apart from being vulnerable to every fingerprinting technique that we introduced in Section 5.4, each extension had one or more of the following issues:

- **Incomplete coverage of the navigator object.** In many cases, while an extension was modifying the `navigator.userAgent` property, it would leave intact other revealing properties of the navigator object, such as `appName`, `appVersion` and `vendor` - Table 5.5. Moreover, the extensions usually left the `navigator.platform` property intact, which allowed for improbable scenarios, like a Microsoft Internet Explorer browser running on Linux.

- **Impossible configurations.** None of the studied extensions attempted to alter the `screen` object. Thus, users who were utilizing laptops or normal workstations and pretended to be mobile devices, were reporting impossible screen width and height (e.g., a reported 1920x1080 resolution for an iPhone).

- **Mismatch between User-agent values.** As discussed earlier, the user-agent of any given browser is accessible through the HTTP headers of a browser request and through the userAgent property of the `navigator` object. We found that some extensions would change the HTTP headers of the browser, but not of the `navigator` object. Two out of three Chrome extensions were presenting this behavior.

We want to stress that these extensions are not malicious in nature. They are legitimately-written software that unfortunately did not account for all possible ways of discovering the true identity of the browsers on which they are installed. The downside here is that, not only fingerprinting libraries can potentially detect the actual identity of a browser, thus, undermining the goals of the extension, but also that they can discover the discrepancies between the values reported by the extensions and the values reported by the browser, and then use these differences as extra features of their fingerprints. The discrepancies of each specific extension can be modeled and thus, as with Adblock Plus, used to uncover the presence of specific extensions, through their side-effects.

The presence of any user-agent-spoofing extension is a discriminatory feature, under the assumption that the majority of browsing users are not familiar enough with privacy threats (with the possible exception of cookies) to install such spoofing extensions. As a rough metric, consider that the most popular extension for Mozilla Firefox is Adblock Plus [1] that, at the time of this writing, is installed by fifteen million users, 25 times more users than UserAgent Switcher, the most popular extension in Table 5.4.

We characterize the extension-problem as an *iatrogenic* [8] one. The users who install these extensions in an effort to hide themselves in a crowd of popular browsers, install software that actually makes them more visible and more distinguishable from the rest of the users, who are using their browsers without modifications. As a result, we advice against the use of user-agent-spoofing extensions as a way of increasing one's privacy. Our findings come in direct antithesis with the advice given by Yen et al. [222], who suggest that user-agent-spoofing extensions *can* be used, as a way of making tracking harder. Even though their study focuses on common identifiers as reported by client-side HTTP headers and the client's IP address, a server capable of viewing these can respond with JavaScript code that will uncover the user-agent-spoofing extension, using any of the aforementioned techniques.

5.6 Discussion

Given the intrusive nature of web-based device fingerprinting and the current inability of browser extensions to actually enhance a user's privacy, in this section, we first discuss possible ways of reducing a user's fingerprintable surface and then briefly describe alternative uses of fingerprinting which may become more prevalent in the future.

5.6.1 Reducing the fingerprintable surface

Flash. As described in Section 5.2, Adobe Flash was utilized by all three fingerprinting libraries that we studied, due to its rich API that allow SWF files to access information not traditionally available through a browser's API. In all cases, the SWF file responsible for gathering information from the host was hidden from the user, by either setting the width and height of the `<object>` tag to zero, or placed into an iframe of zero height and width. In other words, there was no visible change on the web page that included the fingerprinting SWF files. This observation can be used as a first line of defense. All modern browsers have extensions that disallow Flash and Silverlight to be loaded until explicitly requested by the user (e.g., through a click on the object itself). These hidden files cannot be clicked on and thus, will never execute. While this is a straightforward solution that would effectively stop the Flash-part of the fingerprint of all three studied companies, a circumvention of this countermeasure is possible. By wrapping their fingerprinting code into an object of the first-party site and making that object desirable or necessary for the

[8] *iatrogenic* - Of or relating to illness caused by medical examination or treatment.

page's functionality, the fingerprinting companies can still execute their code. This, however, requires much more integration between a first-party website and a third-party fingerprinting company than the current model of "one-size-fits-all" JavaScript and Flash.

In the long run, the best solution against fingerprinting through Flash should come directly from Flash. In the past, researchers discovered that Flash's Local Shared Objects, i.e., Flash's equivalent of browser cookies, were not deleted when a user exited her browser's private mode or even when she used the "Clear Private Data" option of her browser's UI [173]. As a result, in the latest version of Flash, LSOs are not stored to disk but simply kept in memory when the browser's private mode is utilized [220]. Similarly, when a browser enters private mode, Flash could provide less system information, respect any browser-set HTTP proxies and possibly report only a standard subset of a system's fonts, to protect a user's environment from fingerprinting.

JavaScript. There are multiple vendors involved in the development of JavaScript engines, and every major browser is equipped with a different engine. To unify the behavior of JavaScript under different browsers, all vendors would need to agree not only on a single set of API calls to expose to the web applications, but also to internal implementation specifics. For example, hash table implementations may affect the order of objects in the exposed data structures of JavaScript, something that can be used to fingerprint the engine's type and version. Such a consensus is difficult to achieve among all browser vendors, and we have seen diversions in the exposed APIs of JavaScript even in the names of functions that offer the same functionality, e.g., `execScript` and `eval`. Also, based on the fact that the vendors *battle* for best performance of their JavaScript engines, they might be reluctant to follow specific design choices that might affect performance.

At the same time, however, browsers could agree to sacrifice performance when "private-mode" is enabled, where there could be an attempt to expose a unified interface.

5.6.2 Alternative uses of fingerprinting

Although, in this chapter, we have mostly focused on fingerprinting as a fraud-detection and web-tracking mechanism, there is another aspect that requires attention. Drive-by downloads and web attacks in general use fingerprinting to understand if the browser that they are executing on is vulnerable to one of the multiple available exploits. This way, the attackers can decide, at the server-side, which exploit to *reveal* to the client, exposing as little as they can of their attack capabilities. There are three different architectures to detect

drive-by downloads: low-interaction honeypots, high-interaction honeypots and honeyclients. In all three cases, the browser is either a specially crafted one, so that it can instrument the pages visited, or a browser installation that was never used by a real user. Given the precise, browser-revealing, fingerprinting techniques that we described in this chapter, it is possible to see in the future these mechanisms being used by attackers to detect monitoring environments and circumvent detection.

5.7 Related Work

To the best of our knowledge, our work is the first that attempts to study the problem of web-based fingerprinting from the perspectives of all the players involved, i.e., from the perspective of the fingerprinting providers and their fingerprinting methods, the sites utilizing fingerprinting, the users who employ privacy-preserving extensions to combat fingerprinting, and the browser's internals and how they relate to its identity.

Eckersley conducted the first large-scale study showing that various properties of a user's browser and plugins can be combined to form a unique fingerprint [43]. More precisely, Eckersley found that from about 500,000 users who visited `panopticlick.eff.org` and had Flash or Java enabled, 94.2% could be uniquely identified, i.e., there was no other user whose environment produced the same fingerprint. His study, and surprisingly accurate identification results, prompted us to investigate commercial fingerprinting companies and their approach. Yen et al. [222] performed a fingerprinting study, similar to Eckersley's, by analyzing month-long logs of Bing and Hotmail. Interestingly, the authors utilize a client's IP address as part of their tracking mechanism, which Eckersley explicitly avoids dismissing it as "not sufficiently stable." As a way of protecting oneself, the authors advocated the use of user-agent-spoofing extensions. As we discussed in Section 5.5, this is actually counter-productive since it allows for more fingerprinting rather than less.

Mowery et al. [107] proposed the use of benchmark execution time as a way of fingerprinting JavaScript implementations, under the assumption that specific versions of JavaScript engines will perform in a consistent way. Each browser executes a set of predefined JavaScript benchmarks, and the completion-time of each benchmark forms a part of the browser's performance signature. While their method correctly detects a browser-family (e.g., Chrome) 98.2% of the time, it requires over three minutes to fully execute. According to a study conducted by Alenty [54], the average view-time of a web page is 33 seconds. This means that, with high likelihood, the benchmarks will not be able to

completely execute and thus, a browser may be misclassified. Moreover, the reported detection rate of more specific attributes, such as the browser-version, operating system and architecture, is significantly less accurate.

Mowery and Shacham later proposed the use of rendering text and WebGL scenes to a `<canvas>` element as another way of fingerprinting browsers [108]. Different browsers will display text and graphics in a different way, which, however small, can be used to differentiate and track users between page loads. While this method is significantly faster than the execution of browser benchmarks, these technologies are only available in the latest versions of modern browsers, thus they cannot be used to track users with older versions. Contrastingly, the fingerprinting techniques introduced in Section 5.4 can be used to differentiate browsers and their versions for any past version.

Olejnik et al. [133] show that web history can also be used as a way of fingerprinting without the need of additional client-side state. The authors make this observation by analyzing a corpus of data from when the CSS-visited history bug was still present in browsers. Today, however, all modern browsers have corrected this issue and thus, extraction of a user's history is not as straightforward, especially without user interaction [205]. Olejnik et al. claim that large script providers, like Google, can use their near-ubiquitous presence to extract a user's history. While this is true [123], most users have first-party relationships with Google, meaning that they can be tracked accurately, without the need of resorting to history-based fingerprinting.

5.8 Conclusion

In this chapter, we first investigated the real-life implementations of fingerprinting libraries, as deployed by three popular commercial companies. We focused on their differences when compared to Panopticlick and discovered increased use of Flash, backup solutions for when Flash is absent, broad use of Internet Explorer's special features, and the existence of intrusive system-fingerprinting plugins.

Second, we created our own fingerprinting script, using multiple novel features that mainly focused on the differences between special objects, like the `navigator` and `screen`, as implemented and handled by different browsers. We identified that each browser deviated from all the rest in a consistent and measurable way, allowing scripts to almost instantaneously discover the true nature of a browser, regardless of a browser's attempts to hide it. To this end, we also analyzed eleven popular user-agent spoofing extensions and showed

that, even without our newly proposed fingerprinting techniques, all of them fall short of properly hiding a browser's identity.

The purpose of our research was to demonstrate that when considering device identification through fingerprinting, user-privacy is currently on the losing side. Given the complexity of fully hiding the true nature of a browser, we believe that this can be efficiently done only by the browser vendors. Regardless of their complexity and sophistication, browser-plugins and extensions will never be able to control everything that a browser vendor can. At the same time, it is currently unclear whether browser vendors would desire to hide the nature of their browsers, thus the discussion of web-based device fingerprinting, its implications and possible countermeasures against it, must start at a policy-making level in the same way that stateful user-tracking is currently discussed.

Part II

Mitigations for known Web application vulnerabilities

Introduction and Problem Statement

"Distrust and caution are the parents of security."

Benjamin Franklin

In Part I of this thesis, we have explored the workings of specific clusters of web applications and have showed how small implementation choices can lead to large security and privacy problems, specific to certain web applications. For instance, in Chapter 3 we showed how the design choice of using sequential, or otherwise predictable, file identifiers, leads to the leakage of private data from these services to the hands of attackers.

In this second part of this dissertation, we no longer explore services in an attempt to find new vulnerabilities, but rather turn our attention to existing, well-known web application vulnerabilities which are being exploited daily, to attack users and their private data. The common thread that unifies the tackled vulnerabilities is that they all occur due to a server-side misconfiguration or implementation mistake, but they affect the security and privacy of the client-side, i.e, the users of that vulnerable web application. Contrastingly, other popular web application vulnerabilities, such as SQL injection and Remote-File Inclusion, also occur due to the same reasons, but hurt the web application itself, rather than individual users.

Our reaction to these client-side vulnerabilities is to propose, design and implement client-side countermeasures that users can readily deploy transparently, without assistance from the vulnerable web application or reliance to third parties. Thus, we help users protect themselves, by amplifying the security of their browsers regardless of the security provisions of a web application. Moreover, due to the client-side nature of the countermeasures, once one is installed, it will enhance a user's security on all utilized web applications.

More specifically, we propose client-side countermeasures against SSL stripping, session hijacking and malicious, plugin-originating, cross-domain requests.

In Chapter 6, we describe SessionShield, a client-side mechanism that protects a user from session hijacking. SessionShield is based on the observation that session identifiers are strings of data that are intelligible to the web application that issued them but not to the web client which received them. SessionShield removes these identifiers from a user's browser, voiding any session hijacking attacks that may be trying to exfiltrate them. Our system is motivated by the really low adoption of HTTP-Only cookies, a straightforward, client-side browser protection mechanism that needs to be guided by the server-side.

In Chapter 7, we describe a countermeasure against SSL stripping. SSL stripping, proposed by Moxie Marlinspike [98], is an attack where a Man-in-The-Middle (MITM) attacker can suppress the SSL protocol and steal user credentials that would normally be encrypted and exchanged over SSL. We were first in proposing a defense against this attack and our proposed countermeasure takes advantage of a user's browsing habits to automatically create security profiles for the websites that a user visits regularly. Our system detects any future deviation from this profile and warns the users about the website, and network, that they are currently utilizing.

Last, in Chapter 8, we propose DEMACRO, the first countermeasure against the abuse of weak Flash cross-origin request policies. Our work was inspired by three recent studies which all show that many popular web sites can be readily abused by attackers, due to their overly permissive cross-origin policies. DEMACRO identifies Flash-originating requests which, if coupled with unsafe server-side policies, are stripped from their session identifiers. Session identifiers are discovered without the aid of any website, using the techniques presented in Chapter 6.

As with the ecosystems explored in Part I of this dissertation, our work is not exhaustive, in that it does not address all possible client-side attacks. We instead chose to tackle issues for which, either no client-side solutions existed, e.g., SSL stripping, or issues which we felt were not addressed as pragmatically as is expected from a client-side solution.

Chapter 6

Session Hijacking

Preamble

This chapter presents a survey on the adoption of the HTTPOnly cookie mechanism and then proposes a novel client-side protection mechanism which uses various heuristics to identify and isolate session identifiers from the scripting engines situated in browsers. The contents of this chapter are replicated from the paper titled "SessionShield: Lightweight Protection against Session Hijacking" [126], which was published in the proceedings of the 3rd International Symposium on Engineering Secure Software and Systems (ESSoS), in 2011. This work was done with the collaboration of other authors from KU Leuven and SAP Research. Nick Nikiforakis was the lead author of this paper.

6.1 Introduction

Over the past decade, users have witnessed a functional expansion of the Web, where many applications that used to run on the desktop are now accessible through the browser. With this expansion, websites evolved from simple static HTML pages to dynamic Web applications, i.e. content-rich resources accessible through the Web. In this modern Web, JavaScript has proven its usefulness by providing server offloading, asynchronous requests and responses and in general improving the overall user experience of websites. Unfortunately, the de facto support of browsers for JavaScript opened up the user to a new range of attacks,

of which the most common is Cross-site scripting (XSS[1]).

In XSS attacks, an attacker convinces a user's browser to execute malicious JavaScript code on his behalf by injecting this code in the body of a vulnerable webpage. Due to the fact that the attacker can only execute JavaScript code, as opposed to machine code, the attack was initially considered of limited importance. Numerous incidents though, such as the Sammy worm that propagated through an XSS vulnerability on the social network MySpace [167] and the XSS vulnerabilities of many high-impact websites (e.g., Twitter, Facebook and Yahoo [218]) have raised the awareness of the security community. More recently, Apache released information about an incident on their servers where attackers took advantage of an XSS vulnerability and by constant privilege escalation managed to acquire administrator access to a number of servers [217].

Today, the Open Web Application Security Project (OWASP) ranks XSS attacks as the second most important Web application security risk [136]. The Web Hacking Incident Database from the Web Application Security Consortium states that 13.87% of all attacks against Web applications are XSS attacks [203]. These reports, coupled with more than 300,000 recorded vulnerable websites in the XSSed archive [218], show that this problem is far from solved.

In this chapter, we present SessionShield, a lightweight countermeasure against session hijacking. Session hijacking occurs when an attacker steals the session information from a legitimate user for a specific website and uses it to circumvent authentication to that website. Session hijacking is by far the most popular type of XSS attack since every website that uses session identifiers is potentially vulnerable to it. Our system is based on the observation that session identifiers are strings of data that are intelligible to the Web application that issued them but not to the Web client who received them. SessionShield is a proxy outside of the browser that inspects all outgoing requests and incoming responses. Using a variety of methods, it detects session identifiers in the incoming HTTP headers, strips them out and stores their values in its own database. In every outgoing request, SessionShield checks the domain of the request and adds back the values that were previously stripped. In case of a session hijacking attack, the browser will still execute the session hijacking code, but the session information will not be available since the browser never received it. Our system is transparent to both the Web client and the Web server, it operates solely on the client-side and it doesn't rely on the Web server or trusted third parties. SessionShield imposes negligible overhead and doesn't require training or user interaction making it ideal for both desktop and mobile systems.

[1] Cross-site scripting is commonly abbreviated as XSS to distinguish it from the acronym of Cascading Style Sheets (CSS)

6.2 Background

6.2.1 Session Identifiers

The workhorse protocol of the World Wide Web, the HyperText Transfer Protocol (HTTP) and its secure counterpart (HTTP**S**) are by design stateless. That means that a Web application cannot track a client between multiple requests unless it adds a separate tracking mechanism on top of the HTTP(S) protocol. The most commonly used tracking mechanism are sessions identifiers. A session identifier (SID) is a unique string of random data (typically consisting of numbers and characters) that is generated by a Web application and propagated to the client, usually through the means of a cookie. After the propagation of the session, every request initiated by the client will contain, among others, the session identifier that the application entrusted him with. Using session identifiers, the Web application is able to identify individual users, distinguish simultaneously submitted requests and track the users in time. Sessions are used in e-banking, web-mail and virtually every non-static website that needs to enforce access-control on its users. Sessions are an indispensable element of the modern World Wide Web and thus session management support exists in all modern Web languages (e.g., PHP, ASP and JSP).

Session identifiers are a prime attack target since a successful capture-and-replay of such an identifier by an attacker provides him with instant authentication to the vulnerable Web application. Depending on the access privileges of the user whose id was stolen, an attacker can login as a normal or as a privileged user on the website in question and access all sorts of private data ranging from emails and passwords to home addresses and even credit card numbers. The most common way of stealing session identifiers is through Cross-site Scripting attacks which are explained in the following section.

6.2.2 Cross-Site Scripting attacks

Cross-site scripting (XSS) attacks belong to a broader range of attacks, collectively known as code injection attacks. In code injection attacks, the attacker inputs data that is later on perceived as code and executed by the running application. In XSS attacks, the attacker convinces the victim's browser to execute JavaScript code on his behalf thus giving him access to sensitive information stored in the browser. Malicious JavaScript running in the victim's browser can access, among others, the contents of the cookie for the running domain. Since session identifiers are most commonly propagated through cookies, the injected JavaScript can read them and transfer them to an attacker-

controlled server which will record them. The attacker can then replay these sessions to the vulnerable website effectively authenticating himself as the victim.

XSS vulnerabilities can be categorized as *reflected* or *stored*. A reflected XSS vulnerability results from directly including parts of the HTTP request into the corresponding HTTP response. Common examples for reflected XSS issues include search forms that blindly repeat the search term on the results-page or custom 404 error pages. On the other hand, a stored XSS vulnerability occurs whenever the application permanently stores untrusted data which was not sufficiently sanitized. If such data is utilized to generate an HTTP response, all potentially injected markup is included in the resulting HTML causing the XSS issue. Stored XSS was found in the past for instance in guestbooks, forums, or Web mail applications.

Code listing 2 shows part of the code of a search page, written in PHP, that is vulnerable to a reflected XSS attack. The purpose of this page is to read one or more keywords from the user, search the database for the keyword(s) and show the results to the user. Before the actual results, the page prints out the keyword(s) that the user searched for. The programmer however has not provisioned against XSS attacks, and thus whatever is presented as a query, will be “reflected” back in the main page, including HTML and JavaScript code. An attacker can hijack the victim’s session simply by sending him the following link:

```
http://vulnerable.com/search.php?q=</u><script>
document.write('<img src="http://hacker.com/
session_hijack.php?ck=' + document.cookie +'">');
</script>
```

When the user clicks on the above link, his browser will initiate a `GET` request to `vulnerable.com`. The `GET` parameter `q` will be added to the resulting page that the server sends back to the user’s browser. The victim’s browser will start rendering the page and once it reaches the “Search results for:” part, it will create an image URI which contains the values stored in the user’s cookie and ask for that image from the attacker-controlled server. The attacker’s script will record the cookie values and enable the attacker to masquerade as the victim at the `vulnerable.com`.

Listing 2 Code snippet vulnerable to an XSS attack

```
<?php
  session_start();
  ...
  $search_query = $_GET['q'];
  print "Search results for: <u> $search_query </u>";
  ...
?>
```

6.2.3 HTTP-Only and Sessions

Developers realized from early on that it is trivial to hijack Web sessions in the presence of an XSS vulnerability. In 2002, Microsoft developers introduced the notion of `HTTP-Only` cookies and added support for them in the release of Internet Explorer 6, SP1 [104]. `HTTP-Only` is a flag that is sent by the Web application to the client, along with a cookie that contains sensitive information, e.g., a session identifier. It instructs the user's browser to keep the values of that cookie away from any scripting languages running in the browser. Thus, if a cookie is denoted as `HTTP-Only` and JavaScript tries to access it, the result will be an empty string. We tested the latest versions of the five most common Web browsers (Internet Explorer, Firefox, Chrome, Safari and Opera) and we observed that if the Web application emmits the `HTTP-Only` flag the cookie is, correctly, no longer accessible through JavaScript.

In an attempt to discover whether the `HTTP-Only` mechanism is actually used, we crawled the Alexa-ranked top one million websites [5] and recorded whether cookies that contained the keyword "sess" where marked as `HTTP-Only`. We chose "sess" because it is a common substring present in the session names of most major Web languages/frameworks (see Section 6.3.2, Table 6.2) and because of the high probability that customely named sessions will still contain that specific substring. We also provisioned for session names generated by the ASP/ASP.NET framework that don't contain the "sess" string. The results of our crawling are summarized in Table 6.1. Out of 1 million websites, 418,729 websites use cookies in their main page and out of these, 272,335 cookies contain session information[2]. We were surprised to find out that only a 22.3% of all websites containing sessions protected their cookies from session stealing using the `HTTP-Only` method. Further investigation shows that while 1 in 2 ASP websites that use sessions utilize `HTTP-Only`, only 1 in 100 PHP/JSP websites does the same.

[2]Cookies that contained the `HTTP-Only` flag but were not identified by our heurestic are added to the "Other/With HTTP-Only" column.

Session Framework	**Total**	**With HTTP-Only**	**Without HTTP-Only**
PHP	135,117 (53.2%)	1,736 (1.3%)	133,381 (98.7%)
ASP/ASP.NET	60,218 (23.5%)	25,739 (42.7%)	34,479 (57.3%)
JSP	12,911 (5.1%)	113 (0.9%)	12,798 (99.1%)
Other	64,089 (18.2%)	33,071 (51.6%)	31,018 (48.4%)
Total	272,335 (100%)	60,659 (22.3%)	211,676 (77.8%)

Table 6.1: Statistics on the usage of HTTP-Only on websites using session identifiers, sorted according to their generating Web framework

These results clearly show that `HTTP-Only` hasn't received widespread adoption. Zhou et al. [228] recently made a similar but more limited study (top 500 websites, instead of top 1 million) with similar findings. In their paper they acknowledge the usefullness of the `HTTP-Only` mechanism and they discuss possible reasons for its limited deployment.

6.3 SessionShield Design

SessionShield is based on the idea that session identifiers are data that no legitimate client-side script will use and thus should not be available to the scripting languages running in the browser. Our system shares this idea with the `HTTP-Only` mechanism but, unlike `HTTP-Only`, it can be applied selectively to a subset of cookie values and, more important, it doesn't need support from Web applications. This means, that SessionShield will protect the user from session hijacking regardless of the security provisioning of Web operators.

The idea itself is founded on the observation that session identifiers are strings composed by random data and are unique for each visiting client. Furthermore, a user receives a different session identifier every time that he logs out from a website and logs back in. These properties attest that there can be no legitimate calculations done by the client-side scripts using as input the constantly-changing random session identifiers. The reason that these values are currently accessible to client-side scripts is because Web languages and frameworks mainly use the cookie mechanism as a means of transport for the session identifiers. The cookie is by default added to every client request by the browser which aleviates the Web programmers from having to create their own transfer mechanism for session identifiers. JavaScript can, by default, access cookies (using the `document.cookie` method) since they may contain values that the client-side

scripts legitimately need, e.g., language selection, values for boolean variables and timestamps.

6.3.1 Core Functionality

Our system acts as a personal proxy, located on the same host as the browser(s) that it protects. In order for a website or a Web application to set a cookie to a client, it sends a `Set-Cookie` header in its HTTP response headers, followed by the values that it wishes to set. SessionShield inspects incoming data in search for this header. When the header is present, our system analyses the values of it and attempts to discover whether session identifiers are present. If a session identifier is found, it is stripped out from the headers and stored in SessionShield's internal database. On a later client request, SessionShield queries its internal database using the domain of the request as the key and adds to the outgoing request the values that it had previously stripped.

A malicious session hijacking script, whether reflected or stored, will try to access the cookie and transmit its value to a Web server under the attacker's control. When SessionShield is used, cookies inside the browser no longer contain session identifiers and since the attacker's request domain is different from the domain of the vulnerable Web application, the session identifier will not be added to the outgoing request, effectively stopping the session hijacking attack.

In order for SessionShield to protect users from session hijacking it must successfully identify session identifiers in the cookie headers. Our system uses two identification mechanisms based on: a) common naming conventions of Web frameworks and of custom session identifiers and b) statistical characteristics of session identifiers.

6.3.2 Naming Conventions of Session Identifiers

Common Web Frameworks

Due to the popularity of Web sessions, all modern Web languages and frameworks have support for generating and handling session identifiers. Programmers are actually adviced not to use custom session identifiers since their implementation will most likely be less secure from the one provided by their Web framework of choice. When a programmer requests a session identifier, e.g., with `session_start()` in PHP, the underlying framework generates a random unique string and automatically emmits a `Set-Cookie` header containing the generated

Session Framework	Name of Session variable
PHP	phpsessid
ASP/ASP.NET	asp.net_sessionid aspsessionid* .aspxauth* .aspxanonymous*
JSP	jspsessionid jsessionid

Table 6.2: Default session naming for the most common Web frameworks

string in a `name=value` pair, where `name` is a standard name signifying the framework used and `value` is the random string itself. Table 6.2 shows the default names of session identifiers according to the framework used[3]. These naming conventions are used by SessionShield to identify session identifiers in incoming data and strip them out of the headers.

Common Custom Naming

From the results of our experiment in Section 6.2.3, we observed that "sess" is a common keyword among custom session naming and thus it is included as an extra detection method of our system. In order to avoid false-positives we added the extra measure of checking the length and the contents of the value of such a pair. More specifically, SessionShield identifies as session identifiers pairs that contain the word "sess" in their name and their value is more than 10 characters long containing both letters and numbers. These characteristics are common among the generated sessions of all popular frameworks so as to increase the value space of the identifiers and make it practically impossible for an attacker to bruteforce a valid session identifier.

6.3.3 Statistical Characteristics of session identifiers

Despite the coverage offered by the previous mechanism, it is beyond doubt that there can be sessions that do not follow standard naming conventions and thus would not be detected by it. In this part we focus on the fact that session identifiers are long strings of symbols generated in some random way. These two key characteristics, length and randomness, can be used to predict if a

[3]On some versions of the ASP/ASP.NET framework the actual name contains random characters, which are signified by the wildcard symbol in the table.

string, that is present in a cookie, is a session identifier or not. This criterion, in fact, is similar to predicting the strength of a password.

Three methods are used to predict the probability that a string is a session identifier (or equivalently the strength of a password):

1. **Information entropy**: The strength of a password can be measured by the information entropy it represents [52]. If each symbol is produced independently, the entropy is $H = L \cdot \log_2 N$, with N the number of possible symbols and L the length of the string. The resulting value, H, gives the entropy and represents the number of bits necessary to represent the string. The higher the number of necessary bits, the better the strength of the password in the string. For example, a pin-code consisting out of four digits has an entropy of 3.32 bits per symbol and a total entropy of 13.28.

2. **Dictionary check**: The strength of a password reduces if it is a known word. Similarly, cookies that have known words as values are probably not session identifiers.

3. χ^2: A characteristic of a random sequence is that all symbols are produced by a generator that picks the next symbol out of a uniform distribution ignoring previous symbols. A standard test to check if a sequence correlates with a given distribution is the χ^2-test [83], and in this case this test is used to calculate the correlation with the uniform distribution. The less the string is correlated with the random distribution the less probable it is that it is a random sequence of symbols. The uniform distribution used is $1/N$, with N the size of the set of all symbols appearing in the string.

Every one of the three methods returns a probability that the string is a session identifier. These probabilities are combined by means of a weighted average to obtain one final probability. SessionShield uses this value and an experimentally-discovered threshold to differentiate between session and non-session values.

6.4 Evaluation

6.4.1 False Positives and False Negatives

SessionShield can protect users from session hijacking as long as it can successfully detect session identifiers in the incoming HTTP(S) data. In order to evaluate the security performance of SessionShield we conducted the

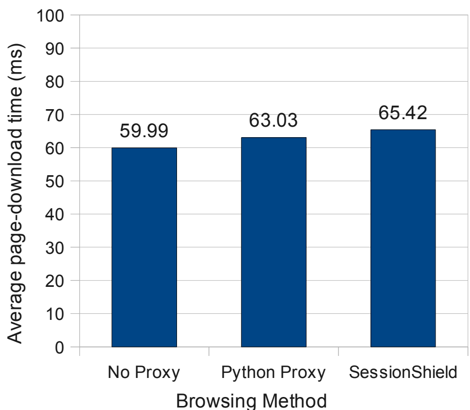

Figure 6.1: Average download time of the top 1,000 websites when accessed locally without a proxy, with a simple forwarding Python-proxy and with SessionShield

following experiment: we seperated the first 1,000 cookies from our experiment in Section 6.2.3 and we used them as input to the detection mechanism of SessionShield. SessionShield processed each cookie and classified a subset of the values as sessions identifiers and the rest as benign data. We manually inspected both sets of values and we recorded the false positives (values that were wrongly detected as session identifiers) and the false negatives (values that were not detected as session identifiers even though they were). SessionShield classified 2,167 values in total (average of 2.16 values/cookie) with 70 false negatives (3%) and 19 false positives (0,8%).

False negatives were mainly session identifiers that did not comply to our session identifier criteria, i.e a) they didn't contain both letters and numbers or b) they weren't longer than 10 characters. Session identifiers that do not comply to these requirements are easily brute-forced even if SessionShield protected them. With regard to false positives, it is important to point out that in order for a website to stop operating correctly under SessionShield, its legitimate client-side scripts must try to use values that SessionShield classified as session identifiers. Thus the actual percentage of websites that wouldn't operate correctly and would need to be white-listed is less than or equal to 0,8%.

6.4.2 Performance Overhead

In an effort to quantify how much would SessionShield change the Web experience of users, we decided to measure the difference in page-download time when a page

is downloaded: a) directly from the Internet; b) through a simple forwarding proxy [64] and c) through SessionShield. Using `wget`, we downloaded the top 1,000 Internet websites [5] and measured the time for each.

In order to avoid network inconsistencies we downloaded the websites locally together with the HTTP headers sent by the actual Web servers. We used a fake DNS server that always resolved all domains to the “loopback” IP address and a fake Web server which read the previously-downloaded pages from disk and replayed each page along with its original headers. This allowed us to measure the time overhead of SessionShield without changing its detection technique, which relies on the cookie-related HTTP headers. It is important to point out that SessionShield doesn’t add or remove objects in the HTML/JavaScript code of each page thus the page-rendering time isn’t affected by its operation. Each experiment was repeated five times and the average page-download time for each method is presented in Fig. 6.1. SessionShield’s average time overhead over a simple Python proxy is approximately 2.5 milliseconds and over a non-proxied environment is 5.4 milliseconds. Contrastingly, popular Web benchmarks show that even the fastest websites have an average page-download time of 0.5 seconds when downloaded directly from the Internet [204].

Since our overhead is two orders of magnitute less than the fastest page-download times we believe that SessionShield can be used by desktop and mobile systems without perceivable performance costs.

6.5 Related Work

Client-side approaches for mitigating XSS attacks: Noxes [81] is a defensive approach closely related to ours – A client-side Web proxy specifically designed to prevent session identifier (SID) theft. Unlike our approach, Noxes does not prevent the injected JavaScript to access the SID information. Instead, Noxes aims to deprive the adversary from the capability to leak the SID value outside of the browser. The proposed technique relies on the general assumption that dynamically assembled requests to external domains are potentially untrustworthy as they could carry stolen SID values. In consequence, such requests are blocked by the proxy. Besides the fact that this implemented policy is incompatible with several techniques from the Web 2.0 world, e.g., Web widgets, the protection provided by Noxes is incomplete: For example, the authors consider static links to external domains to be safe, thus, allowing the attacker to create a subsequent XSS attack which, instead of `script`-tags, injects an HTML-tag which statically references a URL to the attackers domain including the SID value.

Vogt et al. [194] approach the problem by using a combination of static analysis and dynamic data tainting within the browser to track all sensitive information, e.g., SID values, during JavaScript execution. All outgoing requests that are recognised to contain such data are blocked. However, due to the highly dynamic and heterogenous rendering process of webpages, numerous potential hidden channels exist which could lead to undetected information leaks. In consequence, [163] exemplified how to circumvent the proposed technique. In comparison, our approach is immune to threats through hidden channels as the SID never enters the browser in the first place.

Furthermore, browser-based protection measures have been designed that disarm reflected XSS attacks through comparing HTTP requests and responses. If a potential attack is detected, the suspicious code is neutralized on rendering-time. Examples for this approach include NoScript for Firefox [97], Internet Explorer's XSS Filter [160], and XSSAuditor for Chrome [16]. Such techniques are necessarily limited: They are only effective in the presence of a direct, character-level match between the HTTP request and its corresponding HTTP response. All non-trivial XSS vulnerabilities are out of scope. In addition, it is not without risk to alter the HTTP response in such ways: For instance, Nava & Lindsay [117] have demonstrated, that the IE XSS Filter could cause XSS conditions in otherwise secure websites. Finally, to confine potentially malicious scripts, it has been proposed to whitelist trusted scripts and/or to declare untrusted regions of the DOM which disallow script execution [73, 109, 47, 103]. All of these techniques require profound changes in the browser's internals as well as the existence of server-side policy information.

Security enhancing client-side proxies: Besides Noxes, further client-side proxies exist, that were specifically designed to address Web application vulnerabilities: RequestRodeo [78] mitigates Cross-site Request Forgery attacks through selectively removing authentication credentials from outgoing HTTP requests. As discussed in this chapter, the majority of all Web applications utilize the SID as the de facto authentication credential. In consequence, RequestRodeo (and its further refinements, such as [164]) could benefit from our SID detection algorithm (see Section 6.3) in respect to false positive reduction.

Server-side approaches: The majority of existing XSS prevention and mitigation techniques take effect on the Web application's server-side. We only give a brief overview on such related work, as this chapter's contributions specifically address client-side protection: Several approaches, e.g., [142, 120, 62, 221], employ dynamic taint tracking of untrusted data on run-time to identify injection attacks, such as XSS. Furthermore, it has been shown that

static analysis of the application's source code is a capable tool to identify XSS issues (see for instance [93, 216, 80, 201, 55]). Moreover, frameworks which discard the insecure practice of using the string type for syntax assembly are immune against injection attacks through providing suitable means for data/code separation [156, 75]. Jovanovic et al. [79] use a server-side proxy which rewrites HTTP(S) requests and responses in order to detect and prevent Cross-site Request Forgery. Finally, cooperative approaches spanning server and browser have been described in [11, 94, 115].

6.6 Conclusion

Session hijacking is the most common Cross-site Scripting attack. In session hijacking, an attacker steals session-containing cookies from users and utilizes the session values to impersonate the users on vulnerable Web applications. In this chapter we presented SessionShield, a lightweight client-side protection mechanism against session hijacking. Our system, is based on the idea that session identifiers are not used by legitimate client-side scripts and thus shoudn't be available to the scripting engines running in the browser. SessionShield detects session identifiers in incoming HTTP traffic and isolates them from the browser and thus from all the scripting engines running in it. Our evaluation of SessionShield showed that it imposes negligible overhead to a user's system while detecting and protecting almost all the session identifiers in real HTTP traffic, allowing its widespread adoption in both desktop and mobile systems.

Chapter 7

SSL stripping attacks

Preamble

This chapter presents an analysis of SSL stripping attacks, and a client-side countermeasure that can help users detect malicious changes due to a MiTM attacker. The contents of this chapter are replicated from the paper titled "HProxy: Client-side detection of SSL stripping attacks" [129], which was published in the proceedings of the 7th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), in 2010. This work was done with the collaboration of other authors from KU Leuven. Nick Nikiforakis was the lead author of this paper.

7.1 Introduction

In 1994 Netscape Communications released the first complete Secure Sockets Library (SSL) which allowed applications to exchange messages securely over the Internet [177]. This library uses cryptographic algorithms to encrypt and decrypt messages in order to prevent the logging and tampering of these messages by potential eavesdroppers. Today SSL is considered a requirement for companies who handle sensitive user data, such as bank account credentials and credit card numbers. According to a study by Netcraft[119], in January of 2009 the number of valid SSL certificates on the Internet reached one million, recording an average growth of 18,000 certificates per month. Due to its widespread

usage, attackers have developed several attacks, mainly focusing on the forging of invalid SSL certificates and hoping that users will accept them.

Recently however a new attack has surfaced [98]. This technique is not based on any specific programming error but rather on the whole architecture and usage of secure webpages. It is based on the observation that most users never explicitly request SSL protected websites, in the sense that they never type the `https` prefix in their browsers. The transition from cleartext pages to encrypted ones is done usually either through web server redirects, secure links, or secure target links of HTML forms. If an attacker can launch a man-in-the-middle (MITM) attack, he can suppress all such transitions by "stripping" these transitional links from the cleartext HTTP protocol or HTML webpages before forwarding these messages/webpages to the unsuspecting client. Due to stripping of all SSL information, all data that would originally be encrypted are now sent as cleartext by the user's browser providing the attacker with sensitive data such as user credentials to email accounts, bank accounts and credit card numbers used in online transactions.

In this chapter, we explore the idea of using the browser's history as a detection mechanism. We design a client-side proxy which creates a unique profile for each secure website visited by the user. This profile contains information about the specific use of SSL in that website. Using this profile and a set of detection rules, our system can identify when the page has been maliciously altered by a MITM and block the connection with the attacker while notifying the user of an attacker's presence on the network. Our approach does not require server-side cooperation and it does not rely on third-party services.

Our main contributions are:

- Analysis and extension of a new class of web attacks.
- Development of a generic detection ruleset for potential attack vectors.
- Implementation of a client-side proxy which protects end-users from such attacks.

7.2 Anatomy of SSL stripping attacks

Once an attacker becomes MITM on a network, he can modify HTTP messages and HTML elements in order to trick the user's browser into establishing unencrypted connections. In the following two scenarios we present two successful attacks based on redirect suppression and target form re-writing. The

first attack exploits HTTP protocol messages and the second attack rewrites parts of a cleartext HTML webpage.

7.2.1 Redirect Suppression

1. The attacker launches a successful MITM attack against a wireless network becoming the network's gateway. From this point on, all requests and responses from any host on the wireless network are inspected and potentially modified by him.

2. An unsuspecting user from this wireless network uses his browser and types in the URL bar, `mybank.com`. The browser crafts the appropriate HTTP message and forwards the message to the network's gateway.

3. The attacker inspects the message and realizes that the user is about to start a transaction with `mybank.com`. He forwards the message to MyBank's webserver.

4. `mybank.com` protects their entire website using SSL thus, the webserver responds with a 301 (Moved Message) to `https://www.mybank.com`.

5. The attacker intercepts the move message, and instead of forwarding it to the user, he establishes a secure connection with MyBank and after decrypting the resulting HTML, he forwards that to the user.

6. The user's browser receives cleartext HTML, as a response to his request and renders it. What the user now sees is an unencrypted version of MyBank's login page. The only thing that is missing is a subtle lock icon, which would be otherwise located somewhere on the browser window.

7. From this point on, all user-data are transmitted as cleartext to the attacker, where he tunnels them through his own encrypted connection. This results in completely functional but unencrypted web sessions.

7.2.2 Target form re-writing

This attack is quite similar to the redirect suppression attack except for a significant detail. Target form re-writing is an attack against websites which operate mainly over HTTP and they only protect parts of their webpages, such as a login form and any subsequent pages for logged-in users. The way this is constructed in HTML is that while the main page is transferred over HTTP, the target URL of a specific form has an HTTPS prefix. When the

user clicks the "submit" button, the browser recognizes the secure protocol and attempts to establish an SSL connection with the target web server. This is disastrous for an attacker because, even though he controls all local network connections, he has no realistic way of presenting a valid SSL certificate for the secure handshake of the requested web server. The attacker thus, will have to present a self-signed certificate resulting in multiple warnings which the user must accept before proceeding with the connection. In order to avoid this pitfall, the attacker strips all secure form links and replaces them with cleartext versions. So, a form with a target of `https://www.example.com/login.php` becomes `http://www.example.com/login.php` (note the missing `s` from the protocol). The browser has no way of knowing that the original link had a secure target and thus sends the user's credentials over an unencrypted channel. In the same way as before, the attacker uses these credentials in his own valid SSL connection and later forwards to the user the resulting HTML page.

7.3 Effectiveness of the attack

In this section we would like to stress the severity of the SSL attacks described in Section 7.2. We argue that the two main reasons which make SSL stripping such an effective attack are: a) the wide applicability of it in modern networks and b) the way that feedback works on browser software.

7.3.1 Applicability

When eavesdropping attacks were first introduced, they targeted hubbed networks since hubs transmit all packets to all connected hosts, leaving each host to choose the packets that are addressed for itself and disregard the rest. The attacker simply configured his network card to read all packets (promiscuous mode) and had immediate access to all the information coming in and out of the hubbed network. Once hubs started being replaced by switches, this attack was no longer feasible since switches forwarded packets only to the hosts that were intended to receive them (using their MAC addresses as a filter). Attackers had to resort to helper techniques (such as ARP flooding, which filled-up the switch's memory forcing it to start transmitting everything to everyone to keep the network functioning) in order for their eavesdropping attacks to be effective [114].

Today however, due to the widespread use of wireless network connections, attackers have access to hundreds of thousands of wireless networks ranging from home and hotel networks to airport and business networks. Wireless networks

are by definition hubbed networks since the transport medium is “air”. Even certain secure wireless networks, e.g, WEP, are susceptible to MITM attacks as long as the attacker can find the encryption key [199].

The ramifications become even greater when we consider that wireless networks are not restricted to laptops anymore due to the market penetration of hand held devices which use them to connect to the Internet. More and more people use these kind of devices to perform sensitive operations from public wireless networks without suspecting that a potential attacker could be eavesdropping their transactions.

7.3.2 Software feedback

The second main reason that makes this attack effective is that it doesn’t produce negative feedback. Computer users have been unconsciously trained for years that the absence of warning messages and popups means that all operations were successful and nothing unexpected happened. This holds true also for security critical operations where users trust that a webpage is secure as long as the browser remains “silent”.

In the scenario where an attacker tries to present to a web browser a self-signed, expired or otherwise illegal certificate, the browser presents a number of dialogues to the user which inform him of the problems and advise him not to proceed with his request. Modern browsers (such as Firefox) have the user click many times on a number of different dialogues before allowing him to proceed. Many users, understand that it is best to trust their browser’s warnings, especially if they are working from an unfamiliar network (such as a hotel network), even if they end up not doing so [181].

In the SSL stripping attack however, the browser is never presented with any illegal SSL certificates since the attacker strips the whole SSL connection before it reaches the victim. With no warning dialogues, the user has little to no visual cues that something has gone wrong. In the case of SSL-only websites (websites that operate solely under the HTTPS protocol) the only visual cue that such an attack generates is the absence of lock icon somewhere on the browser’s window (something that the attacker can compensate for by changing the .favico icon of the website to a padlock). In partly-protected websites, where the attacker strips the SSL protocol from links and login forms, there are no visual cues and the only way for a user to spot the attack is to manually inspect the source code and identify the parts that have been changed.

7.4 Automatic Detection of SSL stripping

In this section we describe our approach that automatically detects the existence of a MITM attacker conducting an SSL stripping attack on a network. The main strength of MITM attacks is the fact that the attacker has complete control of all data coming in and going out of a network. Any client-side technique trying to detect an attacker's presence must never rely solely on data received by the current network connection.

7.4.1 Core Functionality

Our approach is based on browser history. The observation that lead to this work is that while a MITM attacker has at some point in time, complete control of all traffic on a network, he did not always have this control. We assume that users mainly use secure networks, such as WPA2-protected wireless networks or properly configured switched networks and use insecure networks only circumstantially. Regular browsing of SSL-enabled websites from these secure locations can provide us with enough data to create a profile of what is expected in a particular webpage and what is not.

Our client-side detection tool, History Proxy (HProxy), is trained with the requests and responses of websites that the user regularly visits and builds a profile for each one. It is important to point out that HProxy creates a profile based on the security characteristics of a website and not based on the website's content, enabling it to operate correctly on static as well as most dynamic websites.

HProxy uses the profile of a website, the current browser request and response along with a detection ruleset to identify when a page is maliciously modified by a MITM conducting an SSL stripping attack. The detection ruleset is straightforward and will be explained in detail in Section 7.4.3.

7.4.2 Architecture of HProxy

The architecture of HProxy comprises of the detection ruleset and a number of components which utilize and enforce it - Fig. 7.1. The main components are: a webpage analyzer, which analyzes and identifies the requests initiated from the browser along with the server responses, a MITM Identifier which checks requests and responses against the detection ruleset to decide whether a page is safe or not and lastly a taint module which tries to prevent the leakage of private information even if the MITM-identifier incorrectly tags a page as safe.

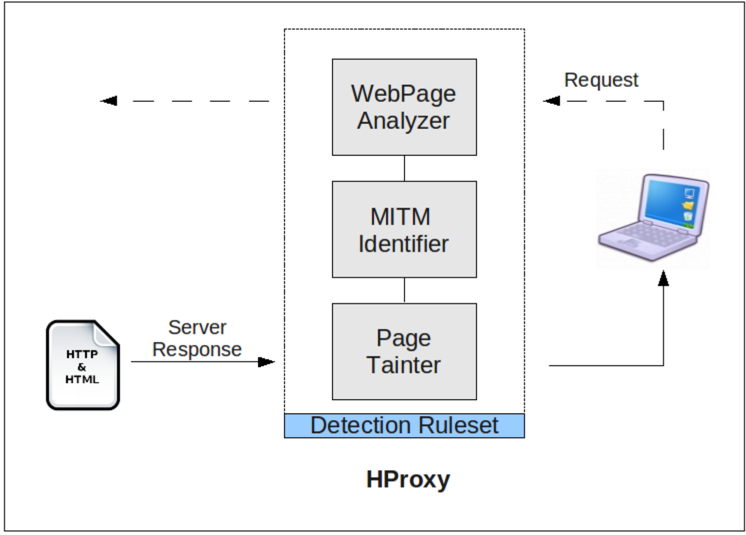

Figure 7.1: Architecture of HProxy

Webpage analyzer

The webpage analyzer is the component responsible of identifying all the critical parts of a webpage. The critical parts of a webpage are the parts that a MITM attacker can insert or alter in order to steal credentials from the end users and are the following:

- JavaScript blocks
- HTTP forms and their targets
- `Iframe` tags
- HTTP Moved messages

The Webpage Analyzer identifies all of the above data structures, along with their attributes and records them in the page's current profile. If a particular page is visited for the first time then this current profile is registered in the profile database, effectively becoming the page's original profile, and the page is forwarded to the user. If not, then the current profile will be checked against the page's original profile by the MITM Identifier. Why these structures are dangerous will be described in detail in Section 7.4.3.

MITM Identifier

The MITM Identifier component encapsulates almost all the detecting capabilities of HProxy (except of the taint component which will be discussed later). It uses the page's current profile as created by the Webpage Analyzer against the page's original profile. In order to make a decision whether a page is altered by an attacker or not, the MITM Identifier utilizes the detection ruleset of HProxy. This ruleset consists of rules for every sensitive data structure that was previously mentioned. Each rule contains the dangerous modifications that can appear in each page, using the page's original profile as a base. Any modifications detected by the Webpage Analyzer that are identifiable by this ruleset are considered a sign of an SSL stripping attack and thus the page is not forwarded to the user.

PageTainter

Even though we have strived to create a ruleset which will be able to detect all malicious modifications we deliberately decided to allow content changes when we cannot decisively classify them as an attack. In order to compensate for these potentially false negatives, HProxy contains a module called PageTainter. The purpose of PageTainter is to enable HProxy to stop in time the leakage of private user data, even when the MITM Identifier module wrongly tags a malicious page as “safe”. For HProxy to stop the leakage of private data, it must first be able to identify what private data is. In order to do this, PageTainter modifies each webpage that contains a secure login form (identifiable by the password-type HTML element) and adds a JavaScript routine which sends the password from it to HProxy once the user types it in. This password is recorded in HProxy in a domain,password tuple[1]. In addition to that, it taints all forms with an extra hidden field which contains location information so that we can later identify which page initiated a GET or a POST request. For each request that initiates from the browser, the PageTainter module, using the hidden domain field checks for the presence of the stored password in the outgoing data. If the page is legitimate, the domain's password will never appear in the HTTP data because it is exchanged only over SSL. A detection of it signifies the fact that an attacker's successful modification passed through our MITM Identifier and is now sending out the password. In this case, HProxy does not allow the connection to be established and informs the user of the attack. To make sure that an attacker will not obfuscate the password beyond recognition

[1]HProxy runs on the same physical host as the browser(s) that it protects thus there are no privacy issues with the stored passwords

by the PageTainter, our detection ruleset has very strict JavaScript rules which will be explained in the next section.

7.4.3 Detection Ruleset

Using the description of SSL-stripping attacks as a base, we studied and recorded all possible HTML and HTTP elements that could be misused by a MITM attacker. This study resulted in a set of pragmatic rules which essentially describe dangerous transitions from the original webpage (as recorded by HProxy) to all future instances of it. A transition can be either an addition of one or more HTML/HTTP elements by the attacker to the original webpage or the modification of existing ones.

The detection ruleset consists of dangerous modifications for every class of sensitive data structures. Each page that comes from the network is checked against each class of rules before it is handed back to the user. In the rest of this section we present the rules for each class of sensitive structures.

HTTP Moved Messages

The HTTP protocol has a variety of protocol messages of which the "moved" messages can be misused in an SSL stripping attack since their suppression can lead to unencrypted sessions (as shown in the example attack in Section 7.2.1). The main rule for this class of messages states that, if the original page profile contains a move message from an `HTTP` to an `HTTPS` page, then any other behavior is potentially dangerous. Given an original request of `HTTP GET` for `domain_a` and an original response stored in the profile database of `MOVED` to `HTTPS domain_a/page_a`, we list all the possible modifications and whether they are allowed by our ruleset, in the following table.

Current Response	**Modification**	**Allowed?**
MOVED HTTPS domain_a/page_a	None	Yes
MOVED HTTPS domain_a/page_b	Changed page	Yes
MOVED HTTP domain_a/page_a	Non-SSL protocol	No
MOVED HTTP domain_b/page_a	Changed domain	No
MOVED HTTPS domain_b/page_a	Changed domain	No
OK `<html>....</html>`	HTML instead of MOVED	No

This ruleset derives from the observation that the developers of a website may decide to create new webpages or rename existing ones, but they will not suddenly stop providing HTTPS nor export their secure service to another

domain. For websites that operate entirely using SSL, this is the only class of rules that will be applied to them as they will operate securely over HTTPS once the MOVE message has been correctly processed.

The rest of the ruleset is there to protect websites that are partly protected by SSL. Such websites use SSL only for their login forms and possibly for the subsequent pages that result after a successful login. The transition from unprotected to protected pages (within the same website) is done usually through a HTTPS form target or through a HTTPS link.

JavaScript

JavaScript is a powerful, flexible and descriptive language that is legitimately used in almost all modern websites to make the user experience better and to offload servers of common tasks that can be executed on the client-side. All these features of JavaScript, including the fact that it is enabled by default in all major browsers make it an ideal target for attackers. Attackers can and have been using JavaScript for a multitude of attacks ranging from Cross-site Scripting [219] to Heap Spraying attacks [174]. For the purpose of stealing credentials, JavaScript can be used to read parts of the webpage (such as a typed-in username and password) and send it out to the attacker.

JavaScript can be categorized as either inline or external. Inline JavaScript, is written inline an HTML webpage, e.g. `<html><script>...</script></html>`. External JavaScript, is written in separate files, present on a webserver that are being included in an HTML page using a special tag, e.g. `<html><script src="http://domain1/js_file.js"> </html>`. Unfortunately for users, both categories of JavaScript can be misused by a MITM. If an attacker adds inline JavaScript in a webpage before forwarding it to the user, the browser has no easy way of discerning which JavaScript parts were legitimately present in the original page and which were later added by the attacker. Also, the attacker can reply to a legitimate external JavaScript request with malicious code since he already has full control over the network and can thus masquerade himself as the webserver.

Because of the nature of JavaScript, HProxy has no realistic way of discerning between original and "added" JavaScript except through the use of whitelisting. The first time that a page which contains an HTTPS form is visited all JavaScript code (internal and external) is identified and recorded in the page's profile. If in a future request of that specific webpage, new or modified JavaScript is identified then the page is tagged as unsafe and it is not forwarded to the user. HProxy's initial whitelisting mechanism involved string comparisons of JavaScript blocks between page loads of the same website. Unfortunately though, the practice

Figure 7.2: Example of an injected HTML form by a MITM attacker

of simple whitelisting can lead to false positives. A way around these false positives is through the use of a JavaScript preprocessor. This preprocessor can distinguish between the JavaScript parts that have been legitimately changed by the web server and the parts which have been added or modified by an attacker. We expand HProxy to include such a preprocessor and we explore this notion in detail later on, in Section 7.5.

Iframe tags can be as dangerous as JavaScript. An attacker can add extra `iframe` tags in order to overlay fake login forms over the real ones [12] or reply with malicious content to legitimate `iframe` requests. Our detection ruleset for `iframe` tags states that no such tags are allowed in pages where an SSL login form is present. The only time an `iframe` tag is allowed is when the original profile of a website states that the login form itself is coded inside the `iframe`.

HTTP Forms can be altered by a MITM attacker so as to prevent the user's browser from establishing an encrypted session with a web server, as was demonstrated in Section 7.2.2. Additionally, extra forms can also be used by an attacker as a way of stealing private information. The set of rules for this class of sensitive data structures is similar to the HTTP Move class ruleset. The previously mentioned Webpage analyzer, records every form, target and protocol for each page that an SSL login form is identified. The ruleset contains the dangerous form modifications that could leak private user credentials. The main rules are applied on the following characteristics:

- **Absence of forms** - The profile for each website maintains information about the number of forms in each page, whether they are login forms and which forms have secure target URLs. Once a missing form is detected, HProxy reads the profile to see the type of the missing form. If the missing

form was a secure login form then HProxy tags this as an attack and drops the request. If the missing form was a plain HTTP form (such as a `Search` form) then HProxy allows the page to proceed.

- **New forms** - New forms can be introduced in a webpage either by web designers (who wish to add functionality to a specific page) or by an attacker who tries to lure the user into typing his credentials in the wrong form - Fig 7.2. If the new form is not a login form then it is an allowed deviation from the page's profile. If the new form is a login-form it is only allowed if the target of the form is secure and in the same domain as the original SSL login form of the page. Even so, there is a chance that a MITM can convince a user to submit his credentials through a non-login form. In these cases, PageTainter will identify the user's password in outgoing data and drop the request before it reaches the attacker.
- **Modified forms** - In this case, an attacker can modify a secure form into an insecure form. Based on the same observation from HTTP moved messages, HProxy does not allow a modified form to be forwarded to the browser if it detects: `(a)` a security downgrade in a login form (the original had an HTTPS target whereas the current one has an HTTP target); or `(b)` a domain change in the target URL

7.4.4 Redirect Suppression Revisited

In Section 7.2.1 we presented one of the most common SSL stripping attacks against browsers, namely redirect suppression. The MITM suppressed the HTTP `Moved` messages and provided the user with an unencrypted version of an originally encrypted website. In this section we repeat the attack but this time, the user is running the HProxy tool. Steps 1-5 are the same with the earlier example but are repeated here for the sake of completeness.

1. The attacker launches a successful MITM attack against a wireless network becoming the network's gateway. From this point on, all requests and responses from any host on the wireless network are inspected and potentially modified by him.
2. An unsuspecting user from this wireless network uses his browser and types in the URL bar, `mybank.com`. The browser crafts the appropriate HTTP message and forwards the message to the network's gateway.
3. The attacker inspects the message and realizes that the user is about to start a transaction with `mybank.com`. He forwards the message to MyBank's webserver.

4. `mybank.com` protects their entire website using SSL thus, the webserver responds with a 301 (Moved Message) to `https://www.mybank.com`.

5. The attacker intercepts the move message, and instead of forwarding it to the user, he establishes a secure connection with MyBank and after decrypting the resulting HTML, he forwards that to the user.

6. HProxy receives the response from the "server" and inspects it. HProxy's trained profile for MyBank states that `mybank.com` is an SSL protected website and when the user requests the website using HTTP, the server redirects him to the HTTPS version of it. This time however HProxy identifies the response as cleartext HTML which is not acceptable according to its detection ruleset.

7. HProxy drops the request and notifies the user about the presence of a MITM on the local network along with specific details.

7.5 Discussion

By analyzing the JavaScript code generated by the top visited websites (as reported by Alexa [5]) we discovered that the dynamic nature of today's Internet doesn't stop in dynamically generated HTML. Many top websites provide different JavaScript code blocks each time they are visited, even when the visits are seconds apart. This means that a simple whitelisting of JavaScript based on string comparison would result in enough false positives to render HProxy unusable. In this section we discuss two techniques that can greatly reduce these false positives: JavaScript preprocessing and Signed JavaScript. The final version of HProxy includes a JavaScript Preprocessor while Signed JavaScript can be used in the future to completely eliminate false positives. We also describe a different way of identifying a MITM by inspecting client requests and the potential problems of that approach.

7.5.1 JavaScript Preprocessing

Most of the JavaScript blocks, even the ones that constantly change, follow a specific structure that can be tracked along page loads. By comparing internal and external JavaScript along two consecutive page loads of a specific webpage, we can discover the static and the dynamic parts of that code. E.g., The JavaScript code in two consecutive loads of Twitter's login page differs only in the contents of a specific variable - Fig. 7.3

We leverage this re-occurring structure to design a JavaScript preprocessor that greatly reduces false positives. When a website is visited for the first time through HProxy, the Webpage Analyzer (Section 7.4.2) makes a duplicate request and compares the JavaScript blocks from the original response and the duplicate one. If the blocks are different it then creates a template of the parts that didn't change and records the place and length of the dynamic parts. This information is stored in the Web pages profile and all future visits of that website will be validated against this template. This enables us, to discern between normal dynamic behavior of a website and JavaScript that was maliciously added by a MITM in order to steal the user's credentials. Although a JavaScript preprocessing that would work on an interpretation level would possibly be able to produce zero false positives we believe that the overhead of such an approach would be prohibitively high and thus we did not research that direction.

7.5.2 Signed JavaScript

Signed JavaScript (SJS) is JavaScript that has been signed by the web server using a valid certificate such as the one used in HTTPS communications. SJS can provide among other features (such as access to restricted JavaScript functions) the guarantee that the script the browser parses has not been modified since it was sent by the Web server [161]. This integrity assurance can be used by HProxy to whitelist unconditionally all JavaScript code blocks that are signed. The downside of this technique is that it requires both server and client-side support[2].

7.5.3 Inspecting Client Requests vs. Server Responses

It is evident that trying to secure JavaScript at the client-side can be a tedious and error-prone process. A different approach of detecting a MITM which may at first appear more appealing is to analyze the client-side requests for anomalous behavior rather than the server-side responses to client-side requests. In such a case, the resulting system would inspect the requests (both secure and insecure) of the browser and compare them to the requests done in the past. A security downgrade of a request, (e.g. the browser is currently trying to communicate to website `X` using an unencrypted channel whereas it always used to communicate over a secure channel), would be a sign of a MITM operating on the network and the request would be dropped. In such a system, JavaScript

[2] At the time of this writing, only Mozilla Firefox appears to support SJS.

```
page.controller_name = 'SessionsController';
page.action_name = 'new';
twttr.form_authenticity_token =
'bcf48ddc78846bea1db1f357300d3e4ad174e2ee';


  page.controller_name = 'SessionsController';
  page.action_name = 'new';
  twttr.form_authenticity_token =
  '644bb1da2eaf04ef5983b7b36d38f411d962856a';
```

Figure 7.3: Portion of the JavaScript code present in two consecutive page loads of the login page of Twitter. The underlined part is the part that changes with each page load

whitelisting would not be an issue since HProxy would only inspect the outgoing requests, regardless of their origin (HTML or JavaScript).

While this approach looks promising it produces more problems than it solves since it has no good way of discerning the nature of new outgoing requests. Consider the scenario where an attacker adds a JavaScript routine which copies the password from the correct form, encrypts it and sends it out using an AJAX request to a new domain. The system would not be able to find a previous outgoing request to match the current request by, and would have to either drop the request (also dropping legitimate new requests - false positives) or let it pass (false negatives). Also, in partly SSL-protected pages, where the client communicates with the same website using both encrypted and unencrypted channels, the MITM could force the browser to send private information over the wrong channel which would again result in leaking credentials.

For these reasons, we decided that a combination of inspecting server responses, preprocessing JavaScript and tracking private data (through the PageTainter - 7.4.2) would be more effective than inspecting client requests and thus we did not implement such a system.

7.6 Evaluation

In this section we provide a security evaluation, the number of false positives and the performance overhead of our approach.

7.6.1 Security Evaluation

HProxy can protect the end-user against the attacks described in [98] as well as a number of new techniques that could be used to steal user credentials in the context of SSL stripping attacks. It can protect the user from credential stealing through redirect suppression, insecure forms, JavaScript methods and injected `iframe` tags.

In order to test the actual effectiveness of our prototype we created a network setup with two clients and a wireless Access Point(AP) with Internet connection. One client was the legitimate user and the other one the MITM, both running the latest version of Ubuntu Linux. From the MITM machine we enabled IP forwarding and we used the `arpspoof` (part of the `dsniff` suite [40]) to position ourselves between the victim machine and the AP. We then run `sslstrip` [178], a tool which strips the SSL links from incoming traffic, creates SSL tunnels with the legitimate websites and captures sensitive data typed by the user. We started browsing the web from the victim machine and we observed that pages which normally are protected through SSL (like GMail and Paypal) were now appearing over HTTP, without any browser warnings whatsoever. Any data typed in fields of those pages were successfully eavesdropped by the MITM host.

We reset the experiment, enabled HProxy and started browsing the web. We browsed through a number of common websites so that HProxy could create a profile for each one of them. We then repeated the procedure of becoming MITM and ran `sslstrip`. Through the victim client, we started visiting all the previously "stripped" websites. This time however, HProxy detected all malicious changes done by `sslstrip` and warned the user of the presence of a MITM attacker on the network.

7.6.2 False Positives

A false positive, is an alert that an Intrusion Detection System (IDS) issues when it detects an attack, that in reality did not happen. When HProxy parses a page, it can occasionally reach to the conclusion that the page was modified by an attacker even if the page was legitimately modified by the web server. These false conclusions can confuse the user as well as undermine his trust of the tool. Most of HProxy's false positives can be generated by its JavaScript rules, as explained in section 7.4.3.

In order to make these occasions as rare as possible we decided to monitor JavaScript blocks only in pages that contain (or originally contained) secure login forms. This decision does not undermine the overall security of HProxy

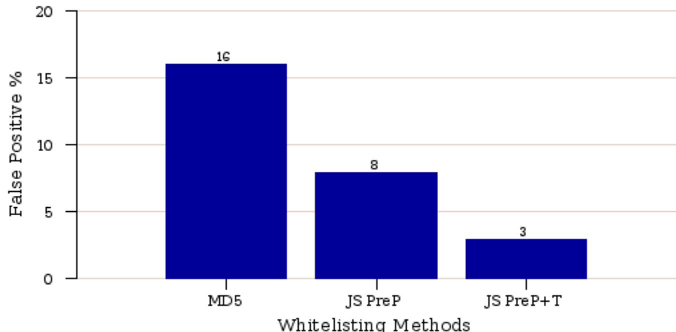

Figure 7.4: False-positive ratio of HProxy using three different methods of whitelisting JavaScript

since in the context of SSL Stripping attacks, JavaScript can only be used to steal credentials as they are typed-in by the user in a secure form. In addition to that, we developed a JavaScript Preprocessor, as explained in Section 7.5.1 which generates a template of each website and a list of expected JavaScript changes.

To measure the amount of false-positives, we compiled a list of 100 websites that contain login pages and we programmed Firefox using `ChickenFoot` [26] to automatically visit them three consecutive times. Firefox's incoming and outgoing traffic was inspected by HProxy which in turn decided whether the page was secure or not. The first time the page was visited, HProxy created a profile for it, which it used for the next two times. Due to our lab's secure network settings, any attack reported by HProxy was a false positive.

In Fig. 7.4 we present the ratio of HProxy's false-positives using three methods of whitelisting JavaScript. The first method that we used is simply gathering all the JavaScript blocks of a webpage and computing their MD5 checksum. If the JavaScript blocks between two page loads differ, then their checksums will also be different. In the second method, we use the JavaScript preprocessor with a strict template, where the changes detected by the preprocessor must be in the precise place and of precise length as the ones originally recorded. Finally we use the same preprocessor but this time we include a "tolerance factor" of 10 characters, where the position and length of changes may vary up to 10 characters (less that 1% of the total length of JavaScript code for most websites).

Using the last method as the whitelisting method of choice, HProxy can handle

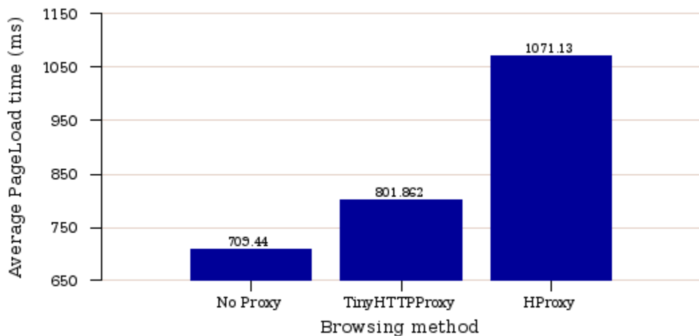

Figure 7.5: Average load time of the top 500 websites of the Internet when accessed locally without a proxy, with a simple forwarding proxy(TinyHTTPProxy) and with HProxy

almost all JavaScript changes successfully. The false-positives are created by webpages which produce JavaScript blocks of different length each time that they are visited. The websites that contain such pages are always the same and can thus be added to a list of unprotected pages.

7.6.3 Performance

To measure the performance overhead of our HProxy prototype, we used a list of the top 500 global websites [5] and we programmed Firefox to visit them ten times each while measuring how much time each page needed to fully load. In order to avoid network inconsistencies we downloaded a copy of each website and browse them locally using a web server that we setup on the same machine that Firefox was running. All caching mechanisms of Firefox were disabled and we were clearing the Linux memory cache between experiments. We repeated the experiment three times and in Fig. 7.5 we present the average load time of Firefox when it run: `(a)` without a proxy `(b)` using a proxy that just forwarded requests to and from Firefox and `(c)` using HProxy. Hproxy shows an overhead of 33% when compared with the forwarding proxy and 51% when compared with Firefox directly accessing the web pages. While this overhead appears substantial, it is important to remember that even the 51% overhead is actually an overhead of 0.41 seconds of time. Firefox starts rendering received content, long before each page fully loads. This means that the user can start "consuming" the content of each page without having to wait for all objects to

be downloaded. Given this behavior, we believe that the added delay of HProxy is only minutely, if at all, perceived by the user during normal web browsing.

7.7 Related work

To the best of our knowledge, our work presents the first countermeasure specifically geared towards SSL stripping attacks. Previous studies mainly focus on the detection of a MITM attacker especially on wireless networks. While a number of these studies detect a wider range of attacks than our approach, it is important to point out that most of them require either specific hardware or knowledge of the network that surpasses the average user's session. This effectively means that unless the techniques are employed before-hand by the administrators of the network they can be of little to no use to the connecting clients. On the other hand HProxy is a client-side tool which protects users from SSL stripping attacks without requiring any support from the wireless network infrastructure.

A number of studies use the information already existing in the 802.11 protocol to identify attackers that try to impersonate legitimate wireless nodes by changing their MAC address. The authors of [61, 214] use the sequence number field of MAC frames as a heuristic for detecting nodes who try to mimic existing MAC addresses. The sequence number is incremented by the node every time that a frame is sent. They create an intrusion detection system which identifies attackers by monitoring invalid, duplicate or dis-proportionally large sequence numbers. Martinez et al. [99] suggest the use of a dedicated passive Wireless Intrusion Detection System (WIDS) which identifies attackers by logging and measuring the time interval between beacon frames. Beacon frames that were broadcasted before the expiration of the last beacon frame (as announced by the AP) are a sign of an impersonation attack. In the same manner, Laroche et. al [90] present a WIDS which uses information such as sequence numbers and fragment numbers, to identify layer-2 attacks. Genetic algorithms are executed against these datasets in an effort to identify impersonating nodes. Unfortunately, their IDS requires training on labeled data sets making it impractical for fast fluctuating wireless networks such as the ones deployed in hotels and airports where wireless nodes are constantly added and removed.

Other researchers have focused more on the physical characteristics of wireless networks and how they relate to intrusion detection. Chen et. al [25] as well as Sheng et al. [170] use the Received Signal Strength (RSS) of a wireless access point as a way to differentiate between the legitimate access point(s) and an

attacker masquerading as one. In both studies, multiple passive gathering devices are used to record the RSS and the data gathered is analyzed using cluster algorithms and Gaussian models. Similarly Suski et al. [182] use special wireless hardware monitors to create and monitor an "RF Fingerprint" based on the inherent emission features of each wireless node. While the detection rates of such studies are quite high, unfortunately their approaches are inherently tied to a significant increase in setup costs (in time, hardware or both) making them unattractive for everyday deployment environments.

Moving up to the top layer of the OSI model, several studies have shown that security systems lack usability and that users accept dialogues and warnings without really understanding the security implications of their actions [6, 37, 53, 181]. Xia et al. [215] try to combat MITM attacks by developing a system which tries to give as much information to the user as possible when invalid certificates are encountered or when a password is about to be transmitted over an unencrypted connection. Due to the nature of SSL stripping attacks, the attacker does not have to present an invalid certificate in order to successfully eavesdrop the user, thus the part of their approach that deals with invalid certificates is ineffective against it. The part that deals with the un-encrypted transmission of a password can be of some use but can be easily circumvented using JavaScript or `iframe` tags as shown in Section 7.4.3.

7.8 Conclusion

Hundreds of thousands of websites rely on SSL daily to protect their customers' traffic from eavesdroppers. Recently though, a new kind of attack against the usage of the SSL protocol surfaced: SSL stripping. The power of such an attack is mainly due the fact that it produces no negative feedback, something that users have been unconsciously trained to search for as an indicator of a page's "insecurity".

In this chapter we argued that SSL stripping attacks are a realistic threat and presented a countermeasure that protects against them. This countermeasure, called HProxy, leverages the browser's history to create security profiles for each website. These profiles contain information about the use of SSL and every future load of that website is validated against that profile. Our prototype implementation of HProxy accurately detected all SSL stripping attacks with very few false positives. Our evaluation of HProxy showed that it can be used with acceptable overhead and without requiring server side support or trusted third parties to secure users against this type of attack.

Chapter 8

Malicious, Plugin-Originating, Cross-domain Requests

Preamble

This chapter presents a client-side countermeasure against malicious cross-origin Flash requests. Our research was motivated by the widespread phenomenon of cross-domain policy misconfigurations, which was witnessed by three recent studies [85, 91, 71]. The contents of this chapter are replicated from the paper titled "DEMACRO: Defense against Malicious Cross-domain Requests" [92], which was published in the proceedings of the 15th International Symposium on Research In Attacks, Intrusions and Defenses (RAID), in 2012. This work was done with the collaboration of other authors from KU Leuven and SAP Research. Nick Nikiforakis was the second author of this paper, being the architect of the countermeasure and evaluating its security and performance overhead.

8.1 Introduction

Since the release of the World Wide Web by CERN, the online world has dramatically changed. In this ever-expanding and ever-changing Web, old technologies give way to new ones with more features enabling developers to constantly enrich their Web applications and provide more content to users.

This evolution of the Web is one of the main reasons that the Internet, once accessible by an elite few, is now populated by almost 2 billion users[1].

Two of the most popular platforms for providing enriched Web content are Adobe Flash and Microsoft Silverlight.Through their APIs, developers can serve data (e.g. music, video and online games) in ways that couldn't be traditionally achieved through open standards, such as HTML. The latest statistics show a 95% and 61% market penetration of Flash and Silverlight respectively, attesting towards the platforms' popularity and longevity [152].

Unfortunately, history and experience have shown that functional expansion and attack-surface expansion go hand in hand. Flash, due to its high market penetration, is a common target for attackers. The last few years have been a showcase of "zero-day" Flash vulnerabilities where attackers used memory corruption bugs to eventually execute arbitrary code on a victim's machine [2].

Apart from direct attacks against these platforms, attackers have devised ways of using legitimate Flash and Silverlight functionality to conduct attacks against Web applications that were previously impossible. One of the features shared by these two platforms is their ability to generate client-side cross-domain requests and fetch content from many remote locations. In general, this is an opt-in feature which requires the presence of a policy configuration. However, in case that a site deploys an insecure wildcard policy, this policy allows adversaries to conduct a range of attacks, such as leakage of sensitive user information, circumvention of CSRF countermeasures and session hijacking. Already, in 2007 a practical attack against Google users surfaced, where the attacker could upload an insecure cross-domain policy file to Google Docs and use it to obtain cross-domain permissions in the rest of Google's services [155]. Even though the security implications of cross-domain configurations are considered to be well understood, three recent studies [85, 91, 71] showed that a significant percentage of websites still utilize highly insecure policies, thus, exposing their user base to potential client-side cross-domain attacks.

To mitigate this threat, we present *DEMACRO*, a client-side defense mechanism which can protect users against malicious cross-domain requests. Our system automatically identifies insecure configurations and reliably disarms potentially harmful HTTP requests through removing existing authentication information. Our system requires no training, is transparent to both the Web server and the user and operates solely on the client-side without any reliance to trusted third-parties.

Our main contributions are:

[1] http://www.internetworldstats.com

- To demonstrate the significance of the topic matter, we provide a practical confirmation of this class of Web application attacks through the analysis of two vulnerable high-profile websites.
- We introduce a novel client-side protection approach that reliably protects end-users against misconfigured cross-domain policies/applets by removing authentication information from potentially malicious situations.
- We report on an implementation of our approach in the form of a Firefox extension called *DEMACRO*. In a practical evaluation we show that *DEMACRO* reliably protects against the outlined attacks while only implying a negligible performance overhead.

8.2 Technical background

In this section we will give a brief overview of client-side cross-domain requests.

8.2.1 The Same-Origin Policy

The Same-Origin Policy (SOP) [162] is the main client-side security policy of the Web. In essence, the SOP enforces that JavaScript running in the Web browser is only allowed access to resources that share the same origin as the script itself. In this context, the origin of a resource is defined by the characteristics of the URL (namely: protocol, domain, and port) it is associated with, hence, confining the capabilities of the script to its *own* application. The SOP governs the access both to local, i.e., within the browser, as well as remote resources, i.e., network locations. In consequence, a JavaScript script can only directly create HTTP requests to URLs that satisfy the policy's same-origin requirements. Lastly, note that the SOP is not restricted to JavaScript since other browser technologies, such as Flash and Silverlight, enforce the same policy.

8.2.2 Client-side Cross-Domain Requests

Despite its usefulness, SOP places limits on modern Web 2.0 functionality, e.g., in the case of Web mash-ups which dynamically aggregate content using cross-domain sources. While in some scenarios the aggregation of content can happen on the server-side, the lack of client-side credentials and potential network restrictions could result in a less-functional and less-personalized mash-up. In order to accommodate this need of fetching resources from multiple sources at

the client-side, Flash introduced the necessary functionality to make controlled client-side cross-domain requests. Following Flash's example, Silverlight and newer versions of JavaScript (using CORS [195]) added similar functionality to their APIs. For the remainder of this chapter we will focus on Flash's approach as it is currently the most wide spread technique [91]. Furthermore, the different techniques are very similar so that the described approach can easily be transferred to these technologies.

8.2.3 An Opt-in Relaxation of the SOP

As we will illustrate in Section 8.3, a general permission of cross-domain requests would result in a plethora of dangerous scenarios. To prevent these scenarios, Adobe designed cross-domain requests as a server-side opt-in feature. A website that desires its content to be fetched by remote Flash applets has to implement and deploy a cross-domain policy which states who is allowed to fetch content from it in a white-list fashion. This policy comes in form of an XML file (`crossdomain.xml`) which must be placed at the root folder of the server (see Listing 3 for an example). The policy language allows the website to be very explicit as to the allowed domains (e.g. `www.a.net`) as well as less explicit through the use of wildcards (e.g. `*.a.net`). Unfortunately the wildcard can be used by itself, in which case all domains are allowed to initiate cross-domain requests and fetch content from the server deploying this policy. While this can be useful in case of well-defined public content and APIs, in many cases it can be misused by attackers to perform a range of attacks against users.

Listing 3 Exemplary crossdomain.xml file

```
<cross-domain-policy>
  <site-control
    permitted-cross-domain-policies="master-only" />
  <allow-access-from domain="a.net"/>
</cross-domain-policy>
```

8.2.4 Client-side cross-domain requests with Flash

Figure 8.1 gives an overview of how Flash conducts client-side cross-domain requests in a legitimate case. (The general scenario is equivalent for Silverlight and only differs in the name and the structure of its policy file). If the domain `a.net` would like to fetch data from the domain `b.net` in the user's

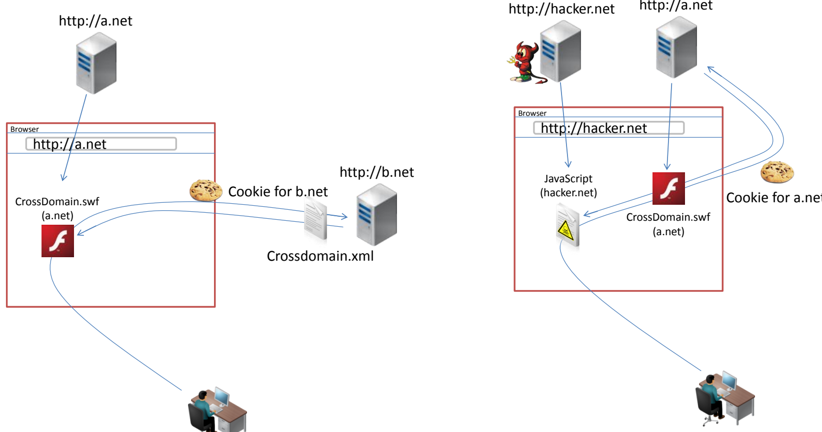

Figure 8.1: General Use Case

Figure 8.2: Vulnerable Flash Proxy

authentication context, it has to include an applet file that implements cross-domain capabilities. This file can either present the fetched data directly or pass it on to JavaScript served by `a.net` for further processing. As already explained earlier, `b.net` has to white-list all domains that are allowed to conduct cross-domain requests. Therefore, `b.net` hosts a cross-domain policy named `crossdomain.xml` in it's root folder. (So the url for the policy-file is `http://b.net/crossdomain.xml`). If the Flash applet now tries to conduct a requests towards `b.net`, the Flash Player downloads the cross-domain policy from `b.net` and checks whether `a.net` is white-listed or not. If so, the request is granted and available cookies are attached to the request. If `a.net` is not white-listed the request is blocked by the Flash Player in the running browser.

8.3 Security Implications of Client-Side Cross-Domain Requests

In this section we present two classes of attacks that can be leveraged by an adversary to steal private data or to circumvent CSRF protection mechanisms.

8.3.1 Vulnerable Scenario 1: Insecure Policy

For this section we consider the same general setup as presented in section 8.2.4. This time however, `b.net` hosts personalized, access-controlled data on its domain and at the same time allows cross-domain requests from any other domain by white-listing a wildcard ("*") in its cross-domain policy. As a result any Flash/Silverlight applet is allowed to conduct arbitrary requests towards `b.net` with the user's session cookie attached to it. Thus, an adversary is able to craft an applet file that can access the personalized, access-controlled data on `b.net`. The last step for the attacker is to lure users into visiting a website that embeds the malicious file. This could either be achieved through social networks, social engineering or through the abuse of other vulnerabilities such as cross-site scripting on vulnerable sites. The more popular the website hosting the insecure policy is, the more chances the attacker has that the users who end up visiting the malicious domain will provide him with authenticated sessions.

8.3.2 Vulnerable Scenario 2: Insecure Flash Proxies

As we have recently shown [77], an insecure cross-domain policy is not the only condition which enables adversaries to conduct the attacks outlined in Section 8.3.1: The second misuse case results from improper use of Flash or Silverlight applets. As stated in Section 8.2.4, an applet is able to exchange data with JavaScript for further processing. For security reasons, communication between JavaScript and Flash/Silverlight applets is also restricted to the same domain. The reason for this is that, as opposed to other embedded content such as JavaScript, embedded Flash files keep their origin. Consequently, JavaScript located on `a.net` cannot communicate with an applet served by `b.net` even if that is embedded in `a.net`. But, as cross-domain communication is also sometimes desirable in this setup, an applet file can explicitly offer communication capabilities to JavaScript served by a remote domain. Therefore, Flash utilizes a white-listing approach by offering the ActionScript directive `System.security.allowDomain(domain)`. With this directive, an applet file can explicitly allow cross-domain communication from a certain domain or white-list all domains by using a wildcard.

We have shown that these wildcards are also misused in practice: Several popular off-the-shelf cross-domain Flash proxies include such wildcard directives, and thus, allow uncontrolled cross-domain JavaScript-to-Flash communication. If such a Flash applet offers cross-domain network capabilities and at the same time grants control over these capabilities to cross-domain JavaScript, an attacker can conduct requests in the name of the website serving the applet file.

Figure 8.2 shows the general setup for this attack. An adversary sets-up a website `hacker.net` that includes JavaScript capable of communicating with a Flash applet served by `a.net`. This applet includes a directive that allows communication from JavaScript served by any other domain. Thus, the attacker is able to instruct the Flash applet to conduct arbitrary requests in the name of `a.net`. If JavaScript from `hacker.net` now conducts a request towards `a.net` via the vulnerable applet, the request itself is not happening cross-domain as `a.net` is the sender as well as the receiver. Therefore, the Flash Player will grant any request without even checking if there is a cross-domain policy in place at `a.net`. Consequently, the attacker can conduct cross-domain requests and read the response as if `a.net` would host a wildcard cross-domain policy. Furthermore, the adversary is also able to misuse existing trust relationships of other domains towards `a.net`. So, if other domains white-list `a.net` in their cross-domain policy, the attacker can also conduct arbitrary cross-domain requests towards those websites by tunneling them through the vulnerable proxy located on `a.net` (please refer to [77] for details concerning this class of attacks).

8.3.3 Resulting malicious capabilities

Based on the presented use and misuse cases we can deduce the following malicious capabilities that an attacker is able to gain.

1. *Leakage of Sensitive Information:* As an adversary is able to conduct arbitrary requests towards a vulnerable website and read the corresponding responses, he is able to leak any information that is accessible via the HTML of that site including information that is bound to the user's session id. Thus, an attacker is able to steal sensitive and private information [60].

2. *Circumvention of Cross-Site Request Forgery Protection:* In order to protect Web applications from cross-site request forgery attacks, many websites utilize a nonce-based approach [23] in which a random and unguessable nonce is included into every form of a Web page. A state changing request towards a website is only granted if a user has requested the form before and included the nonce into the state changing request. The main security assumption of such an approach is that no one else other than the user is able to access the nonce and thus, nobody else is able to conduct state changing requests. As client-side cross-domain requests allow an adversary to read the response of a request, an attacker is able to extract the secret nonce and thus bypass CSRF protections.

3. *Session Hijacking:* Given the fact that an adversary is able to initiate HTTP requests carrying the victim's authentication credentials, he is

essentially able to conduct a session hijacking attack (similar to the one performed through XSS vulnerabilities). As long as the victim remains on the Web page embedding the malicious applet, the attacker can chain a series of HTTP requests to execute complex actions on any vulnerable Web application under the victim's identity. The credentials can be used by an attacker either in an automated fashion (e.g. a standard set of requests towards vulnerable targets) or interactively, by turning the victim in an unwilling proxy and browsing vulnerable Web applications under the victim's IP address and credentials (see Section 8.6.1).

8.3.4 General Risk Assessment

Since the first study on the usage of cross-domain policies conducted by Jeremiah Grossman in 2006 [59], the implementation of cross-domain policies for Flash and Silverlight applets is becoming more and more popular. While Grossman repeated his experiment in 2008 and detected cross-domain policies at 26% of the top 500 websites, the latest experiments show that the adoption of policies for the same set of websites has risen to 52% [91]. Furthermore, the amount of wildcard policies rose from 7% in 2008 up to 11% in 2011. Those figures clearly show that client-side cross-domain requests are of growing importance.

Three recent studies [71, 85, 91] investigated the security implications of cross-domain policies deployed in the wild and all came to the conclusion that cross-domain mechanisms are widely misused. Among the various experiments, one of the studies [91] investigated the Alexa top one million websites and found 82,052 Flash policies, from which 15,060 were found using wildcard policies in combination with authentication tracking and, thus, vulnerable to the range of attacks presented in Section 8.3.

8.4 Real-World Vulnerabilities

To provide a practical perspective on the topic matter, we present in this Section two previously undocumented, real-world cases that show the vulnerabilities and the corresponding malicious capabilities. These two websites are only two examples of thousands of vulnerable targets. However, the popularity and the large user base of these two websites show that even high profile sites are not always aware of the risks imposed by the insecure usage of client-side cross-domain functionality.

8.4.1 Deal-of-the-day Website: Insecure wildcard policy

The vulnerable website features daily deals to about 70 million users world-wide. At the time of this writing, it was ranked on position 309 of the Alexa Top Sites. When we started investigating cross-domain security issues on the website, a crossdomain.xml file was present, which granted any site in the WWW arbitrary cross-domain communication privileges (see Listing 4). This policy can be seen as a worst case example as it renders void all restrictions implied by the Same-Origin Policy and any CSRF protection. On the same domain under which the policy was served, personal user profiles and deal registration mechanisms were available. Hence, an attacker was able to steal any information provided via the HTML user interface. As a proof-of-concept we implemented and tested an exploit which was able to extract private personal information. Furthermore, it was possible to register a user for any deal on the website as CSRF tokens included into every form of the website could be extracted by a malicious Flash or Silverlight applet.

Listing 4 The website's crossdomain.xml file

```
<cross-domain-policy>
  <site-control permitted-cross-domain-policies="all"/>
  <allow-access-from domain="*"/>
  <allow-http-request-headers-from domain="*" headers="*"/>
</cross-domain-policy>
```

8.4.2 Popular sportswear manufacturer: Vulnerable Flash proxy

As discussed in Section 8.3, even without a wildcard cross-domain policy, an attacker is able to conduct arbitrary cross-domain requests under certain circumstances. For this to be possible, a website needs to host a Flash or Silverlight file which is vulnerable to the second misuse case presented in Section 8.3.2.

We found such a vulnerable flash proxy on a Web site of a popular sportswear manufacturer that offers an online store for its products. Although the website's cross-domain policy only includes non-wildcard entries, it hosts a vulnerable Flash proxy which can be misused to circumvent the restrictions implied by the Same-Origin Policy.

Besides leaking private data and circumventing CSRF protections, the vulnerability can be exploited even further by an attacker to misuse existing

trust relationships of the sportswear manufacturer with other websites. As the vulnerable Flash proxy enables an adversary to conduct client-side cross-domain requests in the name of the company, other websites which white-list the sportswear manufacturer's domain in their cross-domain policies are also exposed to attacks. During our tests, we found 8 other websites containing such a white-list entry.

8.5 Client-Side Detection and Mitigation of Malicious Cross-Domain Requests

In Section 8.3.4 we showed that plug-in-based cross-domain techniques are widely used in an insecure fashion and thus users are constantly exposed to risks resulting from improper configuration of cross-domain mechanisms (see Section 8.3 for details). In order to safeguard end-users from these risks we propose *DEMACRO*, a client-side protection mechanism which is able to detect and mitigate malicious plug-in-based cross-domain requests.

8.5.1 High-level Overview

The general mechanism of our approach functions as follows: The tool observes every request that is created within the user's browser. If a request targets a cross-domain resource and is caused by a plugin-based applet, the tool checks whether the request could potentially be insecure. This is done by examining the request's execution context to detect the two misuse cases presented in Section 8.3: For one, the corresponding cross-domain policy is retrieved and checked for insecure wildcards. Furthermore, the causing applet is examined, if it exposes client-side proxy functionality. If one of these conditions is met, the mechanism removes all authentication information contained in the request. This way, the tool robustly protects the user against insecurely configured cross-domain mechanisms. Furthermore, as the request itself is not blocked, there is only little risk of breaking legitimate functionality.

While our system can, in principle, be implemented in all modern browsers, we chose to implement our prototype as a Mozilla Firefox extension and thus the implementation details, wherever these are present, are specific to Firefox's APIs.

8.5.2 Disarming potentially malicious Cross-Domain Requests

A cross-domain request conducted by a plug-in is not necessarily malicious as there are a lot of legitimate use cases for client-side cross-domain requests. In order to avoid breaking the intended functionality but still protecting users from attacks, it is crucial to eliminate malicious requests while permitting legitimate ones. As described in Section 8.3.1 the most vulnerable websites are those that make use of a wildcard policy and host access-controlled, personalized data on the same domain; a practice that is strongly discouraged by Adobe [3]. Hence, we regard this practice as an anti-pattern that carelessly exposes users to high risks. Therefore, we define a potentially malicious request as one that carries access credentials in the form of session cookies or HTTP authentication headers towards a domain that serves a wildcard policy. When the extension detects such a request, it disarms it by stripping session cookies and authentication headers. As the actual request is not blocked, the extension does not break legitimate application but only avoids personalized data to appear in the response.

Furthermore, *DEMACRO* is able to detect attacks against vulnerable Flash proxies as presented in Section 8.3.2. If a page on `a.net` embeds an applet file served by `b.net` and conducts a same-domain request towards `b.net` user credentials are also stripped by our extension. The rationale here is that a Flash-proxy would be deployed on a website so that the website itself can use it rather than allowing any third party domain to embed it and use it.

8.5.3 Detailed Detection Algorithm

While *DEMACRO* is active within the browser it observes any request that occurs. Before applying actual detection and mitigation techniques, *DEMACRO* conducts pre-filtering to tell plugin- and non-plugin-based requests apart. If a plugin-based request is observed, *DEMACRO* needs to check whether the request was caused by a Silverlight or a Flash Applet, in order to download the corresponding cross-domain policy file. With the information in the policy file *DEMACRO* is now able to reveal the nature of a request by assessing the following values:

1. `Embedding Domain:` The domain that serves the HTML document which embeds the Flash or Silverlight file
2. `Origin Domain:` The domain that serves the Silverlight or Flash file and is thus used by the corresponding plug-in as the origin of the request
3. `Target Domain:` The domain that serves the cross-domain policy and is the target for the request

4. `Cross-domain whitelist:` The list of domains (including wildcard entries) that are allowed to send cross-domain requests to the target domain. This information is received either from the Silverlight or Flash cross-domain policy.

Depending on the scenario, the three domains (1,2,3) can either be totally distinct, identical or anything in between. By comparing these values *DEMACRO* is able to detect if a request was conducted across domain boundaries or if a vulnerable proxy situation is present. For the former, the extension additionally checks whether the policy includes a wildcard. If such a potentially malicious situation is detected the extension removes existing HTTP authentication headers or session cookies from the request. Figure 8.3 summarizes our detection algorithm.

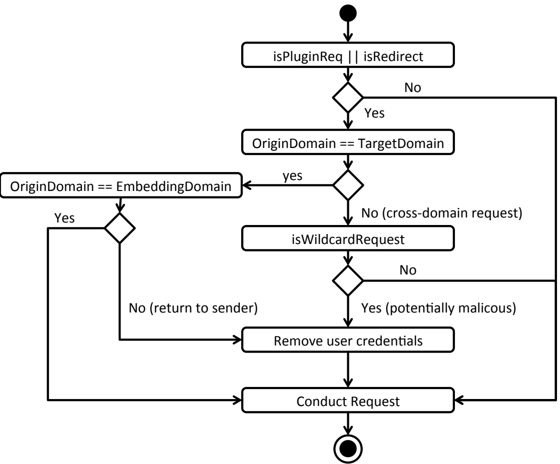

Figure 8.3: Detection and Mitigation Algorithm

In the remainder of this section, we provide technical details how *DEMACRO* handles the tasks of request interception, plugin identification, and session identifier detection.

Requests interception and redirect tracing:

In order to identify plug-in-based requests, *DEMACRO* has to examine each request at several points in time. Firefox offers several different possibilities to intercept HTTP requests, but none of them alone is sufficient for our purpose.

Therefore, we leveraged the capabilities of the `nsIContentPolicy` and the `nsIObserver` interfaces.

The `nsIContentPolicy` interface offers a method called `shouldLoad` which is called each time an HTTP request is initiated and before the actual `HTTPChannel` object is created[2]. Thereby, the method returns a boolean value indicating whether a request should be conducted by Firefox or not. Since we do not want to block a request but only modify its header fields, this method cannot fully serve our purpose. But as it is the only method that receives the url of the webpage and the DOM object that caused the request, we need to intercept page requests here and detect the origin of a request. A request originating from either a `HTMLObjectElement` or from a `HTMLEmbedElement` is categorized as a plug-in-based request.

The `nsIObserver` interface offers the `observe` method which is called at three different points in time:

1. `http-on-modify-request`: Called each time before an HTTP request is sent.
2. `http-on-examine-response`: Called each time before the response is passed back to the caller.
3. `http-on-examine-cache-response`: Called instead of `http-on-examine-response` when the response is completely read from cache.

Thereby, the `observe` method receives an HTTPChannel object as a parameter which can be used to modify request as well as response header fields. If the extension detects a potentially malicious request, it can thus disarm it by stripping existing session information in `Cookie` fields and by removing `Authentication` header fields.

To prevent an attacker from hiding cross-domain requests behind local redirects, the extension also needs to keep track of any redirect resulting from a plug-in-based request. This is also done in the `observe` method at the `http-on-examine-response` event. If a 3xx status code of a plug-in-based request is detected, the redirect location will be stored for examination of follow-up requests.

During experimentation with *DEMACRO* we noticed that Web applications tend to initiate a new session if an existing session identifier is not present in a user's cookie headers. More precisely, if a session identifier never reaches the application, the application emits a `Set-Cookie` header which includes

[2]the HTTPChannel object is used to conduct the request and read the response

Listing 5 `Object` and `Embed` detection (pseudocode)

```
function detectPlugin(HTMLElement elem){

  var type = elem.getAttribute("type");
  var data = elem.getAttribute("data");
  var src  = elem.getAttribute("src");

  switch(type.startsWith){
    case "application/x-silverlight": return flash;
    case "application/x-shockwave-flash": return silverlight;
    default:
  }

  if(data=="data:application/x-silverlight")
    return silverlight;
  if(data.endsWith(".swf")) return flash;

  switch(src.endsWith){
    case ".swf": return flash;
    case ".xap": return silverlight;
    default:
  }

  return -1;
}
```

a new session identifier. If this header reaches the user's browser it will override existing cookies with the same name for the corresponding domain and therefore the user's authenticated session is replaced by an unauthenticated one. As this can obviously lead to undesired side-effects and possible denial of service attacks, *DEMACRO* additionally examines each response of potentially malicious requests and removes `Set-Cookie` headers before allowing the response to be interpreted by the browser.

Plug-in identification:

In order for *DEMACRO* to investigate the correct cross-domain policy, our system must detect whether the underlying request was caused by a Silverlight or by a Flash applet. Since the HTTP request itself does not carry any information about its caller, we developed a mechanism for Firefox to distinguish between Flash and Silverlight requests.

As stated above, the only point in time where we have access to the request-causing DOM element is the call of the `shouldLoad` method in the `nsIContent-Policy` interface. But, due to the fact that Silverlight and Flash files can both be embedded into a page by using either an `HTMLObjectElement` or an `HTMLEmbed-Element`, we need to examine the exact syntax used to embed those files for each element. By testing standard and less-standard ways of embedding an applet to a page, we resulted to the detection mechanism shown in Listing 5. In case the detection mechanism fails, the extension simply requests both policies, in order to prevent an attacker who is trying to circumvent our extension by using an obscure method to embed his malicious files.

Session-Cookie detection:

As described earlier, it is necessary to differentiate between session information and non-session information and strip the former while preserving the latter. The reasoning behind this decision is that while transmitting session identifiers over applet-originating cross-domain requests can lead to attacks against users, non-session values should be allowed to be transmitted since they can be part of a legitimate Web application's logic. *DEMACRO* utilizes the technique described in Chapter 6 in order to identify session identifiers at the client-side.

8.6 Evaluation

8.6.1 Security

In this section we present a security evaluation of *DEMACRO*. In order to test its effectiveness, we used it against MalaRIA [96], a malicious Flash/Silverlight exploit tool that conducts Man-In-The-Middle attacks by having a user visit a malicious website. MalaRIA tunnels attacker's requests through the victim's browser thus making cross-domain requests through the victim's IP address and with the victim's cookies. We chose Joomla, a popular Content Management System, as our victim application mainly due to Joomla's high market penetration [30]. Joomla was installed on a host with a wild-card cross-domain policy, allowing all other domains to communicate with it through cross-domain requests.

Our victim logged in to Joomla and then visited the attacker's site which launched the malicious proxy. The attacker, situated at a different browser, could now initiate arbitrary cross-domain requests to our Joomla installation. Without our countermeasure, the victim's browser added the victim's session

cookie to the outgoing requests, thus authenticating the attacker as the logged-in user. We repeated the experiment with *DEMACRO* activated. This time, the plug-in detected the cross-domain requests and since the Joomla-hosting domain implemented a weak cross-domain policy, it stripped the session-identifier before forwarding the requests. This means that while the attacker could still browse the Joomla website through the victim's browser, he was no longer identified as the logged-in user.

Apart from the security evaluation with existing and publicly available exploits, we also implemented several test-cases of our own. We implemented Flash and Silverlight applets to test our system against all possible ways of conducting cross-domain requests across the two platforms and we also implemented several vulnerable Flash and Silverlight applets to test for the second misuse case (Section 8.3.2). In all cases, *DEMACRO* detected the malicious cross-domain requests and removed the authentication information. Lastly we tested our system against the exploits we developed for the real-world use cases (See Section 8.4) and were able to successfully prevent the attacks in both cases.

8.6.2 Compatibility

In order to test *DEMACRO*'s practical ability to stop potentially malicious cross-domain requests while preserving normal functionality, we conducted a survey of the Alexa top 1,000 websites. We used the Selenium IDE [3] to instrument Firefox to automatically visit these sites twice. The rationale behind the two runs is the following: In the first run, *DEMACRO* was deactivated and the sites and ad banners were populating cookies to our browser. In the second run, *DEMACRO* was enabled and reacting to all the insecure cross-domain requests by stripping-off their session cookies that were placed in the browser during the first run. The results of the second run are summarized in Table 8.1.

In total, we were able to observe 79,093 HTTP requests, of which 1,105 were conducted by plug-ins across domain boundaries. 691 of the requests were considered insecure by *DEMACRO* and thus our system deemed it necessary to remove any session cookies found in these requests. Of the 691, approximately half of them did not contain cookies thus these requests were not modified. For the rest, *DEMACRO* identified at least one session-like value in 275 requests which it removed before allowing the requests to proceed.

In order to find out more about the nature of the insecure requests that *DEMACRO* modified, we further investigated their intended usage: The 275 requests were conducted by a total of 68 domains. We inspected the domains

[3] `http://seleniumhq.org/projects/ide/`

Request Type	**#Requests**
Non-Cross-Domain	77,988 (98.6%)
Safe Cross-Domain	414 (0.52%)
Unsafe Cross-Domain	
Without Cookies	387 (0.49%)
With Session Cookies	275 (0.34%)
With Non-Session Cookies	29 (0.05%)
Total	79,093 (100%)

Table 8.1: Nature of requests observed by DEMACRO for Alexa Top 1k websites

1,500 C.D. requests	**Firefox**	**FF & DEMACRO**	**Overhead/req.**
JavaScript	27.107	28.335	0.00082
Flash	184	210	0.00173

Table 8.2: Best and worst-case microbenchmarks (in seconds) of cross-domain requests

manually and discovered that almost half of the requests where performed by Flash advertising banners and the rest by video players, image galleries and other generic flash applications. We viewed the websites first with *DEMACRO* de-activated and then activated and we noticed that in all but one cases, the applications were loading correctly and displaying the expected content. The one case that did not work, was a Flash advertisement that was no-longer functional when session cookies were stripped away from its requests.

One can make many observations based on the aforementioned results. First of all, we observe that the vast majority of requests do not originate from plugins which experimentally verifies the commonly-held belief that most of the Web's content is served over non-plugin technologies. Another interesting observation is that 50% of the cross-domain plugin-originating requests are towards hosts that implement, purposefully or accidentally, weak cross-domain policies. Finally, we believe that the experiments show that *DEMACRO* can protect against cross-domain attacks without negatively affecting, neither the user's browsing experience nor a website's legitimate content.

8.6.3 Performance

Regardless of the benefits of a security solution, if the overhead that its use imposes is too large, many users will avoid deploying it. In order to evaluate the

performance of our countermeasure we measured the time needed to perform a large number of cross-domain requests when a) issued by JavaScript and b) issued by a Flash applet.

JavaScript Cross-domain requests: This experiment presents the minimum overhead that our extension will add to a user's browser. It consists of an HTML page which includes JavaScript code to fetch 1,500 images from a different domain than the one the page is hosted on. Both domains as well as the browsing user are situated on the same local network to avoid unpredictable network inconsistencies. The requests originating from JavaScript, while cross-domain, are not part of the attack surface explored in this chapter and are thus not inspected by our system. The experiment was repeated 5 times and the first row of Table 8.2 reports the time needed to fetch all 1,500 images with and without our protecting system. The overhead that our system imposes is 0.00082 seconds for each cross-domain request. While this represents the best-case scenario, since none of the requests need to be checked against weak cross-domain policies, we believe that this is very close the user's actual everyday experience where most of the content served is done so over non-plugins and without crossing domain boundaries, as shown in Section 8.6.2.

Flash Cross-domain requests: In this experiment we measure the worst-case scenario where all requests are cross-domain Flash-originating and thus need to be checked and processed by our system. We chose to measure "Flash-Gallery" [4], a Flash-based image gallery that constructs its albums either from images on the local disk of the webserver or using the public images of a given user on `Flickr.com`. Cross-domain accesses occur in the latter case in order for the applet to fetch the necessary information of each image's location and finally the image itself. A feature that made us choose this applet over other Flash-based image galleries is its pre-emptive loading of all available images before the user requests them. Thus, the applet will perform all cross-domain requests needed without any user interaction.

To avoid the network inconsistencies of actually fetching 500 images from Flickr, we implemented the necessary subset of Flickr's protocol to successfully provide a list of image URIs to the Flash applet, in our own Web application which we hosted on our local network. Using our own DNS server, we resolved `Flickr.com` to the host of our Web application instead of the actual Web service. This setup, allowed us to avoid unnecessary modifications on the client-side, i.e. the Flash platform, our plug-in and the Flash applet, and to accurately measure the imposed worst-case overhead of our solution. According to the current protocol of `Flickr.com`, an application first receives a large list of image identifiers. For each identifier, the applet needs to perform 3 cross-domain requests. One to

[4] `http://www.flash-gallery.org/`

receive information about the image, one to fetch the URIs of different image sizes and finally one to fetch the image itself. We configured our Web service to return 500 image identifiers which in total correspond to 1,500 cross-domain requests. Each transfered image had an average size of 128 Kilobytes. Each experiment was repeated 5 times and we report the average timings.

The second row of Table 8.2 reports the results of our experiment. Without any protection mechanisms, the browser fetched and rendered all images in 184 seconds. The reason that made these requests so much slower than the non-protected JavaScript requests of Section 8.6.3 is that this time, the images are loaded into the Flash-plugin and rendered, as part of a functioning interactive image gallery on the user's screen. With our system activated, the same task was accomplished in 210 seconds, adding a 0.00173 seconds overhead to each plugin-based cross-domain request in order to inspect its origin, the policy of the remote-server and finally perform any necessary stripping of credentials.

It is necessary to point out that this overhead represents the upper-bound of overhead that a user will witness in his every-day browsing. In normal circumstances, the majority of requests are not initiated by Flash or Silverlight and thus we believe that the actual overhead will be closer to the one reported in the previous section. Additionally, since our experiments were conducted on the local network, any delay that *DEMACRO* imposes affects the total operation time much more than requests towards remote Web servers where the round-trip time of each request will be significantly larger.

8.7 Related Work

One of the first studies that gave attention to insecure cross-domain policies for Flash, was conducted by Grossman in 2006 [59]. At the time, 36% of the Alexa top 100 websites had a cross-domain policy and 6% of them were using insecure wildcards. Kontaxis et al. [85] recently reported that now more than 60% of the same set of websites implement a cross-domain policy and the percentage of insecure wildcard policies has increased to 21%. While we [91] used a more conservative definition of insecure policies, we also came to the conclusion that the cross-domain traffic through Flash and Silverlight is a real problem.

To the best of our knowledge our work presents the first countermeasure towards this increasingly popular problem. The nature of the problem, i.e. server-side misconfigurations resulting to poor security, allows for two categories of approaches. The first approach is at the server-side, where the administrator of a domain configures the cross-domain policy correctly and thus eliminates the problem all together. While this is the best solution, it a) depends on an

administrator to realize the problem and implement a secure policy and b) needs to be repeated by all administrators in all the domains that use cross-domain policies. Practice has shown that adoption of server-side countermeasures can be a lengthy and often incomplete process [228]. For these reasons we decided to focus our attention on the client-side where our system will protect the user regardless of the security provisions of any given site.

Pure client-side security countermeasures against popular Web application attacks have in general received much attention due to their attractive “install once, secure all” nature. Kirda et al. [81] attempt to stop session hijacking attacks conducted through cross-site scripting (XSS) [219] at the client side using a proxy which blocks requests towards dynamically generated URIs leading to third-party domains. Nikiforakis et al. [126] and Tang et al. [183] tackle the same problem through the identification of session identifiers at the client-side and their subsequent separation from the scripts running in the browser. Vogt et al. [194] also attempt to prevent the leakage of session identifiers through the use of static analysis and dynamic data tainting, however Russo et al. [163] have shown that the identifiers can still be leaked through the use of side channels.

Moving on to Cross-Site Request Forgeries, Johns and Winter [78] propose a solution where a client-side proxy adds tokens in incoming URLs (based on their domains) that bind each URL with their originating domain. At each outgoing request, the domain of the request is checked against the originating domain and if they don’t match, the requests are stripped from their credentials. De Ryck et al. [164] extend this system, by moving it into the browser where more context-information is available. Shahriar and Zulkernine [168] propose a detection technique where each cross-domain request is checked against the visibility of the code that originated it in the user’s browser. According to the authors, legitimate requests will have originated from visible blocks of code (such as a visible HTML form) instead of hidden code (an invisible auto-submitting form or JavaScript code). None of the above authors consider cross-domain requests generated by Flash and Silverlight.

Client-side defense mechanisms have also been used to protect a user’s online privacy. Egele et al. [45] designed a client-side proxy which allows users to make explicit decisions as to which personal information gets transmitted to third-party social network applications. Beato et al. propose a client-side access-control system for social networks, where the publishing user can select who will get access to the published information [19].

8.8 Conclusion

In this chapter we showed that the increasingly popular problem of insecure Flash/Silverlight cross-domain policies is not just an academic problem, but a real one. Even high profile sites carelessly expose their users to unnecessary risks by relying on misconfigured policies and plugin applets. In order to protect security aware users from malicious cross-domain requests we propose a client-side detection and prevention mechanism, *DEMACRO*. *DEMACRO* observes all requests that occur within the user's web browser and checks for potential malicious intent. In this context, we consider a request to be potentially harmful, if it targets a cross-domain resource on a Web server that deploys an insecure wildcard policy. In such a case, *DEMACRO* disarms potentially insecure cross-domain requests by stripping existing authentication credentials. Furthermore, *DEMACRO* is able to prevent the vulnerable proxy attack in which a vulnerable Flash application is misused to conduct cross-domain requests under a foreign identity. We examine the practicality of our approach, by implementing and evaluating *DEMACRO* as a Firefox extension. The results of our evaluation suggest that our system is able to protect against malicious cross-domain requests with a negligible performance overhead while preserving legitimate functionality.

Chapter 9

Conclusion

9.1 Summary

One of the original principles on which the Internet was designed, was the end-to-end principle. This principle allowed the network to take care of the delivery of information without interfering, or attempting to provide any application-specific functionality. This end-to-end nature of the Internet combined with the lack of centralized governance and the ability to deliver an entirely new product, simply through the registration of a domain name and the appropriate acquisition of server resources, allowed people to come-up with products and services never anticipated by the Internet's first architects. It is, at times, hard to grasp that the same, basic protocols that once were merely transferring pages of text from a web server to a primitive web browser, are now allowing entire operating systems and applications to be accessible through one's web browser.

The market created around these products forced the Internet to become mainstream and today, more than a third of the world's entire population has access to the Internet. Many countries have even voted laws that make "access to the Internet", a constitutional right of their citizens. At the same time, it is evident that the majority of Internet users have only a partial understanding of how this technology works.

This ever-growing popularity of the Internet and specifically the web, combined with the users' limited understanding of key security and privacy concepts did not, unfortunately, go by unnoticed by actors with malevolent intent seeking to attack users for financial, political and ideological reasons. Cybercrime arose and attackers took advantage of certain core features of the Internet, like anonymity

and the ability to launch attacks over a distance and across many jurisdictions, in order to make it much harder for officials to track, identify and punish them. As such, cybercrime is much safer for the attacker than real-life crime and in many cases, much more profitable. Moreover, recent events, like the Stuxnet worm attacking Iran's nuclear facilities, changed the way we perceive cybercrime by showing us that cyber attacks can have real, human casualties.

The success of cybercrime combined with the difficulty of prosecuting the responsible actors, prompted many researchers to study each attack instance, as well as the underlying connections of these attacks and to propose tools, techniques and methods to strengthen web applications and their users against them. The industry and government are also beginning to understand that security is not an optional, desired property of systems, but a necessary requirement of an interconnected world.

We believe that the severity of cybercrime and the growing concern of its consequences on the security and privacy of an online world, increases the potential interest and impact of our work which we summarize below. In this dissertation, we focused on two complementary parts related to web security and privacy.

Discovering Critical Threats in Web applications

In the first part, we sought to explore and analyze large online ecosystems, i.e., clusters of similar web applications, from a security and privacy perspective. By uncovering previously unknown vulnerabilities and threats, we showed that conscious and unconscious choices made by the developers of these applications, had the potential to decrease their users' security and invade their privacy. For instance, we demonstrated that the choice of storing files with sequential identifiers in file hosting services, made them and their users vulnerable to file-enumerating attacks, where attackers could obtain private files of users, including pictures, documents and spreadsheets [122]. We also showed that certain conscious choices from web applications developers, like the inclusion of JavaScript code from remote providers [123, 128], and the fingerprinting of their users for fraud-detection and tracking purposes [125], have dire effects on the security and privacy of their users.

Mitigations for known Web application vulnerabilities

In the second part, we focused on protecting the user against well-known attacks. We chose to design and implement client-side countermeasures, as a way of

hardening users against attacks, even if the attacked web applications made no security efforts. For instance, by studying the session management techniques of the modern web, we realized that session identifiers can be of no use to client-side scripts and thus developed heuristics to identify session cookies at the client-side and separate them from the scripting environment of the browser [126]. This technique allowed us to protect the user from session hijacking attacks, and was also re-used to de-authenticate potentially malicious, plugin-originating, client-side requests [92]. Next to session hijacking, we also studied the problem of SSL stripping and proposed the first client-side countermeasure against the attack, that alerts the user of possible SSL stripping attacks using information from previous browsing sessions [129].

9.2 Recent Related Work

Since this dissertation is comprised out of academic papers published during the last three years, there have been related countermeasures that are more recent than each of our contributing publications and thus not covered in the corresponding chapters. In this section, we briefly discuss and compare some major ideas and papers related to the countermeasures presented in Part II of this dissertation.

Session Hijacking Countermeasures

Content Security Policy (CSP) is a countermeasure against XSS and data-injection attacks that was introduced by Mozilla Firefox [109]. In CSP, a web application provides a supporting browser with one or more whitelisted domains, communicated through a new HTTP header. These domains are to be trusted by the browser for remote resources, such as JavaScript code, fonts and images. Any request for remote resources that does not match these domains is discarded, with the optional reporting of the event to the web application. Moreover, CSP, by default, disallows inline JavaScript although a web application can turn this feature off with the use of the “unsafe-inline” directive. While CSP was an experimental feature of Firefox as early as 2009, it only became mainstream in 2012 when its specification was advanced to “Candidate Recommendation” [197], and there was a call for implementations by the W3C [198].

CSP, when enabled and configured properly, is a powerful mechanism which, even though it is not perfect [74], complicates traditional attacks, since the attacker must be able to provide malicious JavaScript through the whitelisted remote domains. The issue with CSP is that it requires a major rewrite

of most web applications since inlining JavaScript in HTML files is a very popular and convenient programming style. In comparison, our session-hijacking countermeasure presented in Chapter 6 is a purely client-side solution which does not require the collaboration of the web application and can thus be used to protect legacy web applications that will never be rewritten to include CSP, or applications that do not *currently* support CSP.

Mulazzani et al. [112] attempt to detect the use of stolen session identifiers at the server-side. The authors utilize browser fingerprinting, done through techniques similar to the ones described in Chapter 5 in order to bind a fingerprint to the session of a legitimate user. The rationale of their approach is the following: If this session identifier is somehow stolen, e.g., through an XSS vulnerability, and used by the attacker to impersonate the user to the vulnerable service, the service will detect the different environment and invalidate the session, thereby stopping the in-progress attack. Even though this makes the use of a stolen session identifier considerably harder, a dedicated attacker can, in most cases, replicate the browsing environment of the victim and thus evade detection at the server-side.

SSL Stripping Countermeasures

HTTP Strict Transport Security (HSTS) is a countermeasure against SSL stripping attacks [150]. In HSTS, a web application instructs a supporting browser that all future requests towards the web application are to be made over SSL. Through the "max-age" directive, a web application can specify the duration of this policy, e.g., `max-age=31536000` instructs the browser to keep this secure policy for one year. Through the use of this countermeasure, a browser never sends an HTTP request in the clear, thus an attacker never gets the opportunity to suppress HTTP to HTTPS transitions, as is the case in SSL stripping attacks.

Similarly with CSP, HSTS requires a collaborating web application in order to be activated. In comparison, HProxy, the SSL stripping countermeasure presented in Chapter 7, is a purely client-side solution. That said, both solutions suffer from the same drawback in that they cannot protect websites that have never been visited by the user, prior the visit done over a network under the control of an active MiTM attacker.

Shin and Lopes [171] propose a system that alerts the user of a potential MiTM conducting an SSL stripping attack, using visual cues. Specifically, they introduce a traffic-light system where HTML input elements are colored red, yellow, and green, depending on the protocol of the surrounding HTML form. For instance, if the destination of a form is over HTTP, the authors' extension

colors all input elements with the red color. The rationale is that a user will recognize “red” as potentially dangerous and not submit the insecure form. While this is an interesting client-side approach, it shifts the problem back to the user who may not be able to reason about the meaning of each color. Moreover, the authors do not consider the cases of malicious JavaScript added by a MiTM, which could either alter the color of each box, or surreptitiously exfiltrate the user’s credentials, regardless of the destination protocol of the form.

9.3 Towards Accurate and Systematic Experiments

In this section, we briefly discuss our choices for the experiments presented throughout our dissertation, and how certain parameters and systematic techniques can be used to ensure better accuracy and reproducibility of similar future experiments.

9.3.1 Storage versus Reproducibility

The web, as a whole, is a very dynamic system. In the span of a few seconds, new domains be registered and placed online, existing long-lived domains may be forcefully taken down, and individual pages of websites can radically change their content. This dynamic nature complicates the reproducibility of experiments that involve “snapshots” of websites. For instance, in Chapter 4, we investigated the practice of remote JavaScript inclusions in the 10,000 most popular websites according to Alexa. In our case, due to storage limitations, we chose to only record each page’s URL and the URLs of all remote scripts, rather than storing the entire page for later analysis. While this was sufficient for our experiments, it limited our ability to reproduce our work and to run additional experiments on the same webpages. Recrawling the same URLs would likely produce slightly different results and thus further complicate correlation of results between crawls. In our case, merely storing the main HTML code from each page would be sufficient, but one can imagine scenarios where the researcher would have to store many more resources, such as all the requests and responses of each website. Thus, for each individual experiment, researchers must anticipate future questions that may arise and store as much data as possible, within their storage limits. This is especially important for long-running, large-scale experiments where stopping and restarting can delay the project for months.

If one takes into account the aforementioned issues and combines them with the fact that hundreds of security researchers across the globe keep on recrawling

the same pages, an opportunity for a testing data-set becomes concrete. We envision the existence of a network where researchers can VPN-into and access the top Internet websites, as crawled by a centralized, neutral service. This setup would allow for repeatable experiments and for the ability to compare findings between research papers, all without the need of rewriting crawlers and data analysis tools.

9.3.2 User Identification

When conducting experiments involving users, a common, desirable measurement is the number of users that reached a service, or performed an action, during a specific window of time. For instance, in Chapter 3, we were interested in identifying the number of malicious users who exploited vulnerable file-hosting services, in order to acquire private user data. In that chapter, and in our dissertation in general, we implicitly treat unique IP addresses as unique users. There have been, however, studies where researchers showed that IP addresses are not an accurate metric for the real number of users, due to the use of DHCP addresses, NAT devices, and proxies [145, 180].

Today, one way that can be used to overcome this problem is user fingerprinting. As we discussed in Chapter 5, certain attributes of a browsing environment can be used to construct a system-specific fingerprint. These techniques, could be used when measuring the number of users reaching an experiment's monitoring infrastructure since they have the ability to differentiate users, even if their requests originate from the same IP address. There are, however, limitations since fingerprinting a user's browsing environment can only be done when there is a browser involved and when attackers use real, JavaScript-enabled, browsers rather than simple, page-fetching tools, like "wget."

9.4 Future Work and Concluding Thoughts

As mentioned in the introduction of this dissertation, the ever-increasing size of the Internet and the speed of innovation in the web and web-related technologies, make it impossible for any single person or organization to explore, in depth, every single attack against the security and privacy of all web applications and their users. Even for the small number of ecosystems that we analyzed, there are still many aspects that require more investigation and thus are candidates for future work.

For instance, while we showed that the way current referrer-anonymizing services are implemented allows abuse from curious advertisers and administrators, we did not investigate what the proper architecture of such services should be, in order to provide users with any guarantees of privacy. In the same way, even though we exposed the ways users are currently fingerprinted by a few commercial fingerprinting companies, there are still many questions that remain unanswered. Further research is necessary to investigate whether there are more fingerprinting companies with more sophisticated methods as well as research into possible ways, both technical and legislative ones, to protect against fingerprinting. At the protection side, our systems identify session identifiers and SSL attacks through sets of efficient, yet imperfect, heuristics, which could be bettered by obtaining and analyzing more data on known session identifiers and variations of session hijacking and SSL stripping attacks.

Overall, we believe that the two parts of this dissertation, work synergetically, exposing new classes of threats against web applications and their users, and proposing ways of defending against already established attacks. We hope that our work will drive new research to tackle the security problems which we uncovered, as well as further refine defensive techniques against well-established security and privacy problems.

Appendix A

Experimental Parameters

In this appendix, we provide certain experimental parameters which were omitted in previous chapters. These parameters are provided so that researchers can experiment with them, compare their selection of parameters to ours and, eventually, arrive to solutions that tackle the sample problems with even better precision and accuracy than that of our own work.

A.1 SessionShield parameters

In Chapter 6, we presented a client-side countermeasure against session hijacking. As part of our countermeasure, we developed a set of tests to automatically measure the "randomness" of each cookie value found in a cookie string. For every value CV_i in a `Set-Cookie` HTTP header, whose key did not match the naming conventions of popular Web frameworks and custom naming practices, we calculated a random score as follows:

$$random_score = \frac{\alpha * encodingSize(CV_i) + \beta * isDictWord(CV_i) + \gamma * chiSquareTest(CV_i)}{\alpha + \beta + \gamma}$$

In our setup, the weights α, β, and γ were assigned the values, 1, 1, and 0.5 respectively. The range of the $random_score$ variable was [0,1] and we empirically chose the value 0.72 as our threshold above which, a cookie value was considered a session identifier and subsequently removed from the incoming HTTP header.

A.2 Quality of Maintenance metric

Detected Mechanism	Points
HTTP Strict-Transport Security	50
Anti-clickjacking	50
Content Security Policy	50
Secure Cookies with Path & Expiration	20
Secure Cookies	10
HTTP-Only Cookies	10
Cache-control: private	10
Pragma: no-cache	10
Anti-XSS	10

Table A.1: Points used, as part of our QoM metric, for enabled, client-side, defense mechanisms and signs of good maintenance

In Chapter 4, we presented a metric for assessing the maintenance quality of JavaScript providers. Our rational was that remote hosts which seem to be ill-maintained may be more vulnerable to attacks that other, well-maintained hosts. Thus, if an administrator decides to include remote JavaScript code, it is more prudent to trust hosts with a higher QoM score.

For any given remote host, our QoM is calculated as follows:

1. We start with the TLS score that SSL Labs assign to remote hosts, based on their SSL/TLS implementation as described in Section 4.3.2. This score ranges from 0 to 100 [176].

2. To the resulting TLS score, we add points for all the client-side defense mechanisms and signs of good maintenance, such as HTTP-Only cookies and cache-control mechanisms, that the remote host enables through the appropriate HTTP headers. The points for each mechanism are listed in Table A.1.

3. Finally, if the web server on the remote host is not up-to-date, then we remove 50 points as a penalty. If the score of a host was already less than 50, then its QoM score resets to 0.

Bibliography

[1] Adblock plus - for annoyance-free web surfing. http://adblockplus.org.

[2] Adobe. Security bulletins and advisories. http://www.adobe.com/support/security/.

[3] Adobe Systems Inc. Cross-domain policy file specification. http://www.adobe.com/devnet/articles/crossdomain_policy_file_spec.html, January 2010.

[4] Aggrawal, 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).

[5] Alexa - Top sites on the Web. http://www.alexa.com/topsites.

[6] Almuhimedi, H., Bhan, A., Mohindra, D., and Sunshine, J. Toward Web Browsers that Make or Break Trust. Poster presented at the fourth Symposium Of Usable Privacy and Security (SOUPS), 2008.

[7] Andersen, A. History of the browser user-agent string. http://webaim.org/blog/user-agent-string-history.

[8] Antoniades, D., Markatos, E. P., and Dovrolis, C. One-click hosting services: a file-sharing hideout. In *Proceedings of the 9th ACM Conference Internet Measurement (IMC)* (2009), pp. 223–234.

[9] Anubis: Analyzing Unknown Binaries. http://anubis.iseclab.org/.

[10] Readability - Arc90 Lab. http://lab.arc90.com/2009/03/02/readability/.

[11] Athanasopoulos, E., Pappas, V., Krithinakis, A., Ligouras, S., Markatos, E. P., and Karagiannis, T. xJS: Practical XSS Prevention for Web Application Development. In *Proceedings of the 1st USENIX*

Conference on Web Application Development (WebApps'10) (2010), pp. 13–13.

[12] Balduzzi, M., Egele, M., Kirda, E., Balzarotti, D., and Kruegel, C. A solution for the automated detection of clickjacking attacks. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security* (2010), ASIACCS '10, pp. 135–144.

[13] Balduzzi, M., Platzer, C., Holz, T., Kirda, E., Balzarotti, D., and Kruegel, C. Abusing social networks for automated user profiling. In *Proceedings of the 13th international conference on Recent Advances in Intrusion Detection* (2010), RAID'10, pp. 422–441.

[14] Barth, A. The Web Origin Concept. `http://tools.ietf.org/html/draft-abarth-origin-09`.

[15] 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.

[16] Bates, D., Barth, A., and Jackson, C. Regular expressions considered harmful in client-side XSS filters. In *Proceedings of the 19th International Conference of the World Wide Web (WWW '10)* (2010), pp. 91–100.

[17] BBC News - Passwords revealed by sweet deal. `http://news.bbc.co.uk/2/hi/technology/3639679.stm`.

[18] BBC News - The web: vital statistics. `http://news.bbc.co.uk/2/hi/technology/8552415.stm`.

[19] Beato, F., Kohlweiss, M., and Wouters, K. Scramble! Your Social Network Data. In *Proceedings of the 11th Privacy Enhancing Technologies Symposium (PETS)* (2011), pp. 211–225.

[20] Beverloo, P. List of Chromium Command Line Switches, –no-referrers. `http://peter.sh/experiments/chromium-command-line-switches`.

[21] Bluecava: Opt out of being tracked. `http://www.bluecava.com/preferences/`.

[22] Bowen, B., Hershkop, S., Keromytis, A., and Stolfo, S. Baiting inside attackers using decoy documents. *Security and Privacy in Communication Networks* (2009), 51–70.

[23] BURNS, J. Cross Site Request Forgery - An introduction to a common web application weakness. Whitepaper, `https://www.isecpartners.com/documents/XSRF_Paper.pdf`, 2005.

[24] CABALLERO, J., GRIER, C., KREIBICH, C., AND PAXSON, V. Measuring Pay-per-Install: The Commoditization of Malware Distribution. In *Proceedings of the the 20th USENIX Security Symposium* (San Francisco, CA, August 2011).

[25] CHEN, Y., TRAPPE, W., AND MARTIN, R. P. Detecting and Localizing Wireless Spoofing Attacks. In *Proceedings of the Fourth Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (IEEE SECON 2007)* (2007), pp. 193–202.

[26] Chickenfoot for Firefox: Rewrite the Web. `http://groups.csail.mit.edu/uid/chickenfoot/faq.html`.

[27] CHU, Z., AND WANG, H. An investigation of hotlinking and its countermeasures. *Computer Communications 34* (April 2011), 577–590.

[28] CLAYTON, R., MURDOCH, S., AND WATSON, R. Ignoring the Great Firewall of China. In *Proceedings of the 6th Privacy Enhancing Technologies Symposium (PETS)* (2006), pp. 20–35.

[29] CLULEY, G. How to turn off Java on your browser - and why you should do it now. `http://nakedsecurity.sophos.com/2012/08/30/how-turn-off-java-browser/`.

[30] Water and Stone: Open Source CMS Market Share Report, 2010.

[31] Collusion: Discover who's tracking you online. `http://www.mozilla.org/en-US/collusion/`.

[32] COMSCORE. The Impact of Cookie Deletion on Site-Server and Ad-Server Metrics in Australia, January 2011.

[33] COVA, M., KRUEGEL, C., AND VIGNA, G. Detection and analysis of drive-by-download attacks and malicious JavaScript code. In *Proceedings of the 19th International World Wide Web Conference (WWW)* (2010), pp. 281–290.

[34] DE GROEF, W., DEVRIESE, D., NIKIFORAKIS, N., AND PIESSENS, 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), pp. 748–759.

[35] De Ryck, P., Desmet, L., Joosen, W., and Piessens, F. Automatic and precise client-side protection against CSRF attacks. In *Proceedings of the 16th European conference on Research in Computer Security (ESORICS)* (September 2011), pp. 100–116.

[36] Devriese, D., and Piessens, F. Noninterference Through Secure Multi-Execution. In *Proceedings of the IEEE Symposium on Security and Privacy* (2010), pp. 109–124.

[37] Dhamija, R., Tygar, J. D., and Hearst, M. Why phishing works. In *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems* (2006), pp. 581–590.

[38] Dingledine, R., Mathewson, N., and Syverson, P. Tor: The Second-Generation Onion Router. In *Proceedings of the 13th USENIX Security Symposium* (2004), pp. 303 – 320.

[39] Dong, X., Tran, M., Liang, Z., and Jiang, X. AdSentry: comprehensive and flexible confinement of JavaScript-based advertisements. In *Proceedings of the 27th Annual Computer Security Applications Conference* (2011), ACSAC '11, pp. 297–306.

[40] dsniff. `http://monkey.org/~dugsong/dsniff/`.

[41] ebricca. Firefox extension : refspoof. `https://addons.mozilla.org/en-US/firefox/addon/refspoof/`.

[42] Eckersley, P. Panopticlick | Self-Defense. `https://panopticlick.eff.org/self-defense.php`.

[43] Eckersley, P. How Unique Is Your Browser? In *Proceedings of the 10th Privacy Enhancing Technologies Symposium (PETS)* (2010), pp. 1–18.

[44] ECMAScript Language Specification, Standard ECMA-262, Third edition.

[45] Egele, M., Moser, A., Kruegel, C., and Kirda, E. Pox: Protecting users from malicious facebook applications. In *Proceedings of the 3rd IEEE International Workshop on Security in Social Networks (SESOC)* (2011), pp. 288 –294.

[46] Endler, D. Brute-Force Exploitation of Web Application Session IDs. *Retrieved from http://www. cgisecurity. com* (2001), 1–40.

[47] Erlingsson, U., Livshits, B., and Xie, Y. End-to-end Web Application Security. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS'07)* (May 2007).

[48] Recent referer xss vulnerabilities. `http://evuln.com/xss/referer.html`.

[49] Falliere, N., Murchu, L. O., and Chien, E. W32.Stuxnet Dossier, February 2011.

[50] Fioravanti, M. Client fingerprinting via analysis of browser scripting environment, 2010.

[51] Mozilla Firefox : Add-on Statistics . `https://addons.mozilla.org/en-US/statistics/addons_in_use/`.

[52] Florencio, D., and Herley, C. A large-scale study of web password habits. In *Proceedings of the 16th International World Wide Web Conference (WWW '07)* (New York, NY, USA, 2007), pp. 657–666.

[53] Friedman, B., Hurley, D., Howe, D. C., Felten, E., and Nissenbaum, H. Users' conceptions of web security: a comparative study. In *CHI '02: CHI '02 extended abstracts on Human factors in computing systems* (2002), pp. 746–747.

[54] Gassée, J.-L., and Filloux, F. Measuring Time Spent On A Web Page. `http://www.cbsnews.com/2100-215_162-5037448.html`.

[55] Geay, E., Pistoia, M., Tateishi, T., Ryder, B., and Dolby, J. Modular String-Sensitive Permission Analysis with Demand-Driven Precision. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)* (2009), pp. 177–187.

[56] Ghostery. `http://wwww.ghostery.com`.

[57] Chromium Security :: Vulnerability Rewards Program. `http://www.chromium.org/Home/chromium-security/vulnerability-rewards-program`.

[58] Grier, C., Ballard, L., Caballero, J., Chachra, N., Dietrich, C. J., Levchenko, K., Mavrommatis, P., McCoy, D., Nappa, A., Pitsillidis, A., Provos, N., Rafique, M. Z., Rajab, M. A., Rossow, C., Thomas, K., Paxson, V., Savage, S., and Voelker, G. M. Manufacturing Compromise: The Emergence of Exploit-as-a-Service. In *Proceedings of the 19th ACM Conference on Computer and Communication Security (CCS)* (October 2012).

[59] Grossman, J. crossdomain.xml statistics. `http://jeremiahgrossman.blogspot.com/2006/10/crossdomainxml-statistics.html`.

[60] Grossman, J. I used to know what you watched, on YouTube. [online], http://jeremiahgrossman.blogspot.com/2008/09/i-used-to-know-what-you-watched-on.html, Accessed in January 2011, September 2008.

[61] Guo, F., and cker Chiueh, T. Sequence Number-Based MAC Address Spoof Detection. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID)* (2005), pp. 309–329.

[62] Halfond, W. G., Orso, A., and Manolios, P. Using Positive Tainting and Syntax-Aware Evaluation to Counter SQL Injection Attacks. In *Proceedings of the 14th ACM Symposium on the Foundations of Software Engineering (FSE)* (2006), pp. 175–185.

[63] Hickson, I., and Hyatt, D. HTML 5 Working Draft - The sandbox Attribute. http://www.w3.org/TR/html5/the-iframe-element.html#attr-iframe-sandbox, June 2010.

[64] Hisao, S. Tiny HTTP Proxy in Python. http://www.okisoft.co.jp/esc/python/proxy/.

[65] Homepage. Privoxy. http://www.privoxy.org.

[66] I2P Anonymous Network. http://www.i2p2.de/.

[67] ietf-http-wg mailinglist. Re: Referer: (sic) from Phillip M. Hallam-Baker on 1995-03-09. http://lists.w3.org/Archives/Public/ietf-http-wg-old/1995JanApr/0109.html.

[68] Jakobsson, M., Finn, P., and Johnson, N. Why and How to Perform Fraud Experiments. *Security & Privacy, IEEE 6*, 2 (March-April 2008), 66–68.

[69] Jakobsson, M., and Ratkiewicz, J. Designing ethical phishing experiments: a study of (ROT13) rOnl query features. In *15th International World Wide Web Conference (WWW)* (2006), pp. 513–522.

[70] 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.

[71] Jang, D., Venkataraman, A., Swaka, G. M., and Shacham, H. Analyzing the Cross-domain Policies of Flash Applications. In *Proceedings of the 5th Workshop on Web 2.0 Security and Privacy (W2SP)* (2011).

[72] Jensen, C. CryptoCache: a secure sharable file cache for roaming users. In *Proceedings of the 9th workshop on ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system* (2000), vol. 54, pp. 73–78.

[73] Jim, T., Swamy, N., and Hicks, M. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In *Proceedings of the 16th International World Wide Web Conference (WWW '07)* (May 2007), pp. 601–610.

[74] Johns, M. PreparedJS: Secure Script-Templates for JavaScript. In *Proceedings of the 10th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)* (2013), pp. 102–121.

[75] Johns, M., Beyerlein, C., Giesecke, R., and Posegga, J. Secure Code Generation for Web Applications. In *Proceedings of the 2nd International Symposium on Engineering Secure Software and Systems (ESSoS '10)* (2010), pp. 96–113.

[76] Johns, M., Engelmann, B., and Posegga, J. XSSDS: Server-side detection of cross-site scripting attacks. In *Proceedings of the 24th Annual Computer Security Applications Conference (ACSAC)* (2008), pp. 335–344.

[77] Johns, M., and Lekies, S. Biting the hand that serves you: A closer look at client-side flash proxies for cross-domain requests. In *Proceedings of the 8th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)* (2011), pp. 85–103.

[78] Johns, M., and Winter, J. RequestRodeo: Client Side Protection against Session Riding. In *Proceedings of the OWASP Europe 2006 Conference* (2006).

[79] Jovanovic, N., Kirda, E., and Kruegel, C. Preventing cross site request forgery attacks. In *Proceedings of IEEE International Conference on Security and Privacy for Emerging Areas in Communication Networks (Securecomm)* (2006), pp. 1–10.

[80] Jovanovic, N., Kruegel, C., and Kirda, E. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In *Proceedings of the 27th IEEE Symposium on Security and Privacy (IEEE S&P)* (2006), pp. 258–263.

[81] 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.

[82] Klein, A. How Fraudsters are Disguising PCs to Fool Device Fingerprinting. `http://www.trusteer.com/blog/how-fraudsters-are-disguising-pcs-fool-device-fingerprinting`.

[83] Knuth, D. E. *The Art of Computer Programming, Volume 2*. Addison-Wesley Publishing Company, 1971.

[84] Kolbitsch, C., Livshits, B., Zorn, B., and Seifert, C. Rozzle: De-cloaking internet malware. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (IEEE S&P)* (May 2012), pp. 443–457.

[85] Kontaxis, G., Antoniades, D., Polakis, I., and Markatos, E. P. An empirical study on the security of cross-domain policies in rich internet applications. In *Proceedings of the 4th European Workshop on Systems Security (EUROSEC)* (2011).

[86] Krebs, B. How to Unplug Java from the Browser. `http://krebsonsecurity.com/how-to-unplug-java-from-the-browser`.

[87] Krishnamurthy, B. Privacy leakage on the Internet. presented at IETF 77, March 2010.

[88] Krishnamurthy, B., and Wills, C. E. Generating a privacy footprint on the Internet. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement* (New York, NY, USA, 2006), IMC '06, pp. 65–70.

[89] Lai, E. What's replacing P2P, BitTorrent as pirate hangouts? `http://www.computerworld.com/s/article/9139210/`.

[90] LaRoche, P., and Zincir-Heywood, A. N. Genetic Programming Based WiFi Data Link Layer Attack Detection. In *Proceedings of the 4th Annual Communication Networks and Services Research Conference (CNSR '06)* (2006), pp. 285–292.

[91] Lekies, S., Johns, M., and Tighzert, W. The state of the cross-domain nation. In *Proceedings of the 5th Workshop on Web 2.0 Security and Privacy (W2SP)* (2011).

[92] Lekies, S., Nikiforakis, N., Tighzert, W., Piessens, F., and Johns, M. DEMACRO: Defense against Malicious Cross-domain Requests. In *Proceedings of the 15th International Symposium on Research In Attacks, Intrusions and Defenses (RAID)* (2012), pp. 254–273.

[93] Livshits, B., and Lam, M. S. Finding Security Vulnerabilities in Java Applications Using Static Analysis. In *Proceedings of the 14th USENIX Security Symposium* (August 2005), pp. 18–18.

[94] Louw, M. T., and Venkatakrishnan, V. BluePrint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers. In *Proceedings of the 30th IEEE Symposium on Security and Privacy (IEEE S&P)* (May 2009), pp. 331–346.

[95] Magazinius, J., Phung, P. H., and Sands, D. Safe wrappers and sane policies for self protecting JavaScript. In *Proceedings of the 15th Nordic conference on Information Security Technology for Applications* (2010), NordSec '10, pp. 239–255.

[96] Malaria - i'm in your browser, surfin your webs. `http://erlend.oftedal.no/blog/?blogid=107`, 2010.

[97] Maone, G. NoScript Firefox Extension, 2006.

[98] Marlinspike, M. New Tricks for Defeating SSL in Practice. In *Proceedings of BlackHat 2009* (DC, 2009).

[99] Martínez, A., Zurutuza, U., Uribeetxeberria, R., Fernández, M., Lizarraga, J., Serna, A., and Vélez, I n. Beacon Frame Spoofing Attack Detection in IEEE 802.11 Networks. In *ARES '08: Proceedings of the 2008 Third International Conference on Availability, Reliability and Security* (2008), pp. 520–525.

[100] Mayer, J. R. Tracking the Trackers: Early Results | Center for Internet and Society. `http://cyberlaw.stanford.edu/node/6694`.

[101] Mayer, J. R. Any person... a pamphleteer. Senior Thesis, Stanford University, 2009.

[102] Mayer, J. R., and Mitchell, J. C. Third-party web tracking: Policy and technology. In *IEEE Symposium on Security and Privacy* (2012), pp. 413–427.

[103] Meyerovich, L., and Livshits, B. ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser. In *Proceedings of the 31st IEEE Symposium on Security and Privacy (IEEE S&P)* (May 2010), pp. 481–496.

[104] Microsoft. Mitigating Cross-site Scripting With HTTP-only Cookies.

[105] Mills, E. Device identification in online banking is privacy threat, expert says. CNET News (April 2009).

[106] Moore, T., and Edelman, B. Measuring the perpetrators and funders of typosquatting. In *Proceedings of the 14th international conference on Financial Cryptography and Data Security* (Berlin, Heidelberg, 2010), FC'10, Springer-Verlag, pp. 175–191.

[107] Mowery, K., Bogenreif, D., Yilek, S., and Shacham, H. Fingerprinting information in JavaScript implementations. In *Proceedings of the Web 2.0 Security & Privacy Workshop (W2SP 2011)* (2011).

[108] Mowery, K., and Shacham, H. Pixel perfect: Fingerprinting canvas in HTML5. In *Proceedings of the Web 2.0 Security & Privacy Workshop (W2SP 2011)* (2012).

[109] Mozilla Foundation. Content Security Policy Specification. `https://wiki.mozilla.org/Security/CSP/Spec`, 2009.

[110] MozillaZine. Network.http.sendRefererHeader - MozillaZine Knowledge Base. `http://kb.mozillazine.org/Network.http.sendRefererHeader`.

[111] Mueller, W. Top free file hosts to store your files online. `http://www.makeuseof.com/tag/top-free-file-hosts/`.

[112] Mulazzani, M., Unger, T., Weippl, E., Schrittwieser, S., Huber, M., and Frühwirt, D. Shpf: Enhancing http(s) session security with browser fingerprinting. In *Proceedings of the Eighth International Conference on Availability, Reliability and Security (ARES)* (9 2013).

[113] Murphey, L. Secure session management: Preventing security voids in web applications, 2005.

[114] Nachreiner, C. Anatomy of an ARP Poisoning Attack. `http://www.watchguard.com/infocenter/editorial/135324.asp`.

[115] Nadji, Y., Saxena, P., and Song, D. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In *Network & Distributed System Security Symposium (NDSS '09)* (2009).

[116] Narayanan, A., and Shmatikov, V. Robust de-anonymization of large sparse datasets. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy (IEEE S&P)* (2008), pp. 111–125.

[117] Nava, E. V., and Lindsay, D. Our favorite XSS filters/IDS and how to attack them. Presentation at the BlackHat US conference, 2009.

[118] Netcraft. March 2012 Web Server Survey. `http://news.netcraft.com/archives/2012/03/05/march-2012-web-server-survey.html`.

[119] Netcraft. One Million SSL Sites on the Web. `http://news.netcraft.com/archives/2009/02/01/one_million_ssl_sites_on_the_web.html`.

[120] Nguyen-Tuong, A., Guarnieri, S., Greene, D., Shirley, J., and Evans, D. Automatically hardening web applications using precise tainting. In *Proceedings of the 20th IFIP International Information Security Conference* (2005), pp. 295–308.

[121] Nikiforakis, N., Acker, S. V., Meert, W., Desmet, L., Piessens, F., and Joosen, W. Bitsquatting: Exploiting bit-flips for fun, or profit? In *Proceedings of the 22nd International World Wide Web Conference (WWW)* (2013), pp. 989–998.

[122] Nikiforakis, N., Balduzzi, M., Van Acker, S., Joosen, W., and Balzarotti, D. Exposing the lack of privacy in file hosting services. In *Proceedings of the 4th USENIX conference on Large-scale exploits and emergent threats* (2011), LEET'11.

[123] Nikiforakis, N., Invernizzi, L., Kapravelos, A., Acker, S. V., Joosen, W., Kruegel, C., Piessens, F., and Vigna, G. You Are What You Include: Large-scale Evaluation of Remote JavaScript Inclusions. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2012), pp. 736–747.

[124] Nikiforakis, N., Joosen, W., and Johns, M. Abusing Locality in Shared Web Hosting. In *Proceedings of the 4th European Workshop on System Security (EuroSec)* (2011), pp. 2:1–2:7.

[125] 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* (2013), pp. 541–555.

[126] 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.

[127] Nikiforakis, N., Piessens, F., and Joosen, W. HeapSentry: Kernel-assisted Protection against Heap Overflows. In *Proceedings of the 10th Conference on Detection of Intrusions and Malware & Vulnerability Analysis (DIMVA)* (2013), pp. 177–196.

[128] Nikiforakis, N., Van Acker, S., Piessens, F., and Joosen, W. Exploring the Ecosystem of Referrer-Anonymizing Services. In *Proceedings of the 12th Privacy Enhancing Technology Symposium (PETS)* (2012), pp. 259–278.

[129] 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 & Vulnerability Assessment (DIMVA)* (2010), pp. 200–218.

[130] Norton. Cybercrime report, 2011. http://now-static.norton.com/now/en/pu/images/Promotions/2012/cybercrime/assets/downloads/en-us/NCR-DataSheet.pdf.

[131] Noyes, D. Top 15 Valuable Facebook Statistics. http://zephoria.com/social-media/top-15-valuable-facebook-statistics/.

[132] Ocariza Jr., F., Pattabiraman, K., and Zorn, B. Javascript errors in the wild: An empirical study. In *Proceedings of the 22nd International Symposium on Software Reliability Engineering (ISSRE)* (2011), pp. 100 –109.

[133] Olejnik, Ł., Castelluccia, C., and Janc, A. Why Johnny Can't Browse in Peace: On the Uniqueness of Web Browsing History Patterns. In *the 5th workshop on Hot Topics in Privacy Enhancing Technologies (HOTPETS)* (2012).

[134] Opera. Disabling referrer logging - Opera Knowledge Base. http://www.opera.com/support/kb/view/93/.

[135] OWASP. Cross-site Scripting (XSS). https://www.owasp.org/index.php/XSS.

[136] OWASP Top 10 Web Application Security Risks. http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project.

[137] Passeri, P. List Of Hacked Celebrities Who Had (Nude) Photos Leaked. http://hackmageddon.com/2012/08/07/list-of-hacked-celebrities-who-had-nude-photos-leaked/.

[138] Payment Card Industry. (Approved Scanning Vendor) Program Guide. https://www.pcisecuritystandards.org/pdfs/asv_program_guide_v1.0.pdf.

[139] Pearson, S. Taking account of privacy when designing cloud computing services. In *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing* (2009), CLOUD '09, pp. 44–52.

[140] Phung, P. H., Sands, D., and Chudnov, A. Lightweight self-protecting JavaScript. In *Proceedings of the 4th International Symposium*

on Information, Computer, and Communications Security (New York, NY, USA, 2009), ASIACCS '09, ACM, pp. 47–60.

[141] Pierson, G., and DeHaan, J. Patent US20080040802 - NETWORK SECURITY AND FRAUD DETECTION SYSTEM AND METHOD.

[142] Pietraszek, T., and Berghe, C. V. Defending against Injection Attacks through Context-Sensitive String Evaluation. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID)* (2005), pp. 124–145.

[143] Provos, N. A virtual honeypot framework. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13* (2004), SSYM'04, pp. 1–1.

[144] Qtip compromised. `https://github.com/Craga89/qTip2/issues/286`.

[145] Rajab, M. A., Zarfoss, J., Monrose, F., and Terzis, A. My botnet is bigger than yours (maybe, better than yours): why size estimates remain challenging. In *Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets* (2007), HotBots'07, pp. 5–5.

[146] Ratanaworabhan, P., Livshits, B., and Zorn, B. G. JSMeter: comparing the behavior of JavaScript benchmarks with real web applications. In *Proceedings of the 2010 USENIX conference on Web application development* (2010), WebApps'10, pp. 3–3.

[147] Reis, C., Barth, A., and Pizano, C. Browser Security: Lessons from Google Chrome. *Queue 7*, 5 (June 2009), 3:3–3:8.

[148] Reis, C., Dunagan, J., Wang, H. J., Dubrovsky, O., and Esmeir, S. BrowserShield: vulnerability-driven filtering of dynamic HTML. In *OSDI '06: Proceedings of the 7th symposium on Operating Systems Design and Implementation* (2006), pp. 61–74.

[149] Reiter, M. K., and Rubin, A. D. Crowds: anonymity for web transactions. *ACM Transactions on Information and System Security (TISSEC) 1* (November 1998), 66–92.

[150] RFC 6797 - HTTP Strict Transport Security (HSTS). `http://tools.ietf.org/html/rfc6797`.

[151] RFC 2616 - Hypertext Transfer Protocol – HTTP/1.1. `http://tools.ietf.org/html/rfc2616`.

[152] Rich internet application (ria) market share. `http://www.statowl.com/custom_ria_market_penetration.php`.

[153] Richards, G., Hammer, C., Burg, B., and Vitek, J. The eval that men do: A large-scale study of the use of eval in javascript applications. In *Proceedings of the 25th European conference on Object-oriented programming* (Berlin, Heidelberg, 2011), ECOOP'11, Springer-Verlag, pp. 52–78.

[154] Richards, G., Lebresne, S., Burg, B., and Vitek, J. An analysis of the dynamic behavior of javascript programs. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2010), PLDI '10, ACM, pp. 1–12.

[155] Rios, B. B. Cross domain hole caused by google docs. `http://xs-sniper.com/blog/Google-Docs-Cross-Domain-Hole/`.

[156] Robertson, W., and Vigna, G. Static Enforcement of Web Application Integrity Through Strong Typing. In *Proceedings of the USENIX Security Symposium* (Montreal, Canada, August 2009).

[157] Robertson, W., Vigna, G., Kruegel, C., and Kemmerer, R. A. Using generalization and characterization techniques in the anomaly-based detection of web attacks. In *Proceedings of the 13th Symposium on Network and Distributed System Security (NDSS)* (2006).

[158] Roesner, F., Kohno, T., and Wetherall, D. Detecting and defending against third-party tracking on the web. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI)* (2012), pp. 12–12.

[159] Ross, D. IE8 Security Part IV: The XSS Filter. `http://blogs.msdn.com/b/ie/archive/2008/07/02/ie8-security-part-iv-the-xss-filter.aspx`.

[160] Ross, D. IE 8 XSS Filter Architecture/Implementation, August 2008.

[161] Ruderman, J. JavaScript Security: Signed Scripts. `http://www.mozilla.org/projects/security/components/signed-scripts.html`.

[162] Ruderman, J. The Same Origin Policy. `http://www.mozilla.org/projects/security/components/same-origin.html`, August 2001.

[163] Russo, A., Sabelfeld, A., and Chudnov, A. Tracking Information Flow in Dynamic Tree Structures. In *14th European Symposium on Research in Computer Security (ESORICS)* (2009), pp. 86–103.

[164] Ryck, P. D., Desmet, L., Heyman, T., Piessens, F., and Joosen, W. CsFire: Transparent Client-Side Mitigation of Malicious Cross-Domain Requests. In *Proceedings of 2nd International Symposium on Engineering Secure Software and Systems (ESSoS '10)* (2010), pp. 18–34.

[165] Saponas, T. S., Lester, J., Hartung, C., Agarwal, S., and Kohno, T. Devices that tell on you: privacy trends in consumer ubiquitous computing. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium* (2007), pp. 5:1–5:16.

[166] Scott, J. How many Firefox users have add-ons installed? 85%! https://blog.mozilla.org/addons/2011/06/21/firefox-4-add-on-users/.

[167] Security, W. XSS Worms: The impending threat and the best defense.

[168] Shahriar, H., and Zulkernine, M. Client-side detection of cross-site request forgery attacks. In *Proceedings of the IEEE 21st International Symposium on Software Reliability Engineering (ISSRE)* (2010), pp. 358 –367.

[169] Sharky. 100 of the best free file hosting upload sites. http://filesharefreak.com/2009/08/26/100-of-the-best-free-file-hosting-upload-sites/.

[170] Sheng, Y., Tan, K., Chen, G., Kotz, D., and Campbell, A. Detecting 802.11 MAC Layer Spoofing Using Received Signal Strength. In *Proceedings of INFOCOM 2008* (2008), pp. 1768 – 1776.

[171] Shin, D., and Lopes, R. An empirical study of visual security cues to prevent the ssl stripping attack. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC)* (2011), pp. 287–296.

[172] Solove, D. J. 'I've Got Nothing to Hide' and Other Misunderstandings of Privacy. In *San Diego Law Review* (2007), vol. 44.

[173] Soltani, A., Canty, S., Mayo, Q., Thomas, L., and Hoofnagle, C. J. Flash Cookies and Privacy. In *SSRN preprint* (2009).

[174] Sotirov, A. Heap Feng Shui in Javascript. In *Proceedings of BlackHat Europe 2007* (2007).

[175] Spafford, E. H. The internet worm program: An analysis. *Computer Communication Review 19* (1988).

[176] SSL Labs Server Rating Guide. https://www.ssllabs.com/downloads/SSL_Server_Rating_Guide_2009.pdf.

[177] The SSL Protocol. http://www.webstart.com/jed/papers/HRM/references/ssl.html.

[178] Moxie Marlinspike's sslstrip. http://www.thoughtcrime.org/software/sslstrip/.

[179] 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* (New York, NY, USA, 2010), WWW '10, ACM, pp. 921–930.

[180] Stone-Gross, B., Cova, M., Cavallaro, L., Gilbert, B., Szydlowski, M., Kemmerer, R., Kruegel, C., and Vigna, G. Your botnet is my botnet: analysis of a botnet takeover. In *Proceedings of the 16th ACM conference on Computer and communications security (CCS)* (2009), pp. 635–647.

[181] Sunshine, J., Egelman, S., Almuhimedi, H., Atri, N., and Cranor, L. F. Crying Wolf: An Empirical Study of SSL Warning Effectiveness. In *Proceedings of the 18th Usenix Security Symposium* (2009), pp. 399–416.

[182] Suski, W., Temple, M., Mendenhall, M., and Mills, R. Using Spectral Fingerprints to Improve Wireless Network Security. In *Global Telecommunications Conference, 2008. IEEE GLOBECOM 2008. IEEE* (30 2008-Dec. 4 2008), pp. 1–5.

[183] Tang, S., Dautenhahn, N., and King, S. T. Fortifying web-based applications automatically. In *Proceedings of the 8th ACM Conference on Computer and Communications Security* (2011), pp. 615–626.

[184] 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* (Aug. 2010), pp. 371–388.

[185] The New York Times - John Schwartz. Giving the Web a Memory Cost Its Users Privacy. http://www.nytimes.com/2001/09/04/technology/04COOK.html.

[186] The Wall Street Journal. What They Know. http://blogs.wsj.com/wtk/.

[187] Thierry Zoller. How NOT to implement a Payback/Cashback System. In *OWASP BeNeLux* (2010).

[188] Tor Project: Anonymity Online. http://www.torproject.org.

[189] Torbutton: I can't view videos on YouTube and other flash-based sites. Why? https://www.torproject.org/torbutton/torbutton-faq.html.en#noflash.

[190] Turow, J., King, J., Hoofnagle, C. J., Bleakley, A., and Hennessy, M. Americans Reject Tailored Advertising and Three Activities that Enable It, September 2009.

[191] Ur, B., Leon, P. G., Cranor, L. F., Shay, R., and Wang, Y. Smart, useful, scary, creepy: perceptions of online behavioral advertising. In *Proceedings of the Eighth Symposium on Usable Privacy and Security* (2012), SOUPS '12, pp. 4:1–4:15.

[192] Van Acker, S., De Ryck, P., Desmet, L., Piessens, 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* (2011), ACSAC '11, pp. 307–316.

[193] VirusTotal - Free Online Virus, Malware and URL Scanner. https://www.virustotal.com/.

[194] Vogt, P., Nentwich, F., Jovanovic, N., Kruegel, C., Kirda, E., and Vigna, G. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS '07)* (2007).

[195] W3C. Cross-Origin Resource Sharing. http://www.w3.org/TR/cors/.

[196] W3C Member Submission: Web Tracking Protection. http://www.w3.org/Submission/2011/SUBM-web-tracking-protection-20110224/.

[197] W3C : Content Security Policy 1.0. http://www.w3.org/TR/CSP/, 2012.

[198] W3C Invites Implementations of Content Security Policy 1.0. http://www.w3.org/News/2012#entry-9633, 2012.

[199] Walker, J. R., Walker, S. P. J., and Corporation, I. Unsafe at any key size; An analysis of the WEP encapsulation, 2000.

[200] Wang, Y.-M., Beck, D., Wang, J., Verbowski, C., and Daniels, B. Strider typo-patrol: discovery and analysis of systematic typo-squatting. In *Proceedings of the 2nd conference on Steps to Reducing Unwanted Traffic on the Internet - Volume 2* (2006), SRUTI'06, pp. 5–5.

[201] Wassermann, G., and Su, Z. Static Detection of Cross-Site Scripting Vulnerabilities. In *Proceedings of the 30th International Conference on Software Engineering* (Leipzig, Germany, 2008), pp. 171–180.

[202] Wayback Machine. http://archive.org.

[203] Web Application Security Consortium. Web Hacking Incident Database. http://projects.webappsec.org/Web-Hacking-Incident-Database.

[204] Performance Benchmark - Monitor Page Load Time | Webmetrics.

[205] 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 2011 IEEE Symposium on Security and Privacy* (2011), SP '11, pp. 147–161.

[206] West, R. The psychology of security. *Communications of the ACM 51*, 4 (Apr. 2008), 34–40.

[207] WHATWG. HTML - Living standard. http://www.whatwg.org/specs/web-apps/current-work/multipage/links.html.

[208] Wikipedia. Referrer spam. http://en.wikipedia.org/wiki/Referrer_spam.

[209] Wilander, J., Nikiforakis, N., Younan, Y., Kamkar, M., and Joosen, W. RIPE: Runtime Intrusion Prevention Evaluator. In *In Proceedings of the 27th Annual Computer Security Applications Conference, (ACSAC)* (2011), pp. 41–50.

[210] Wondracek, G., Holz, T., Kirda, E., and Kruegel, C. A practical attack to de-anonymize social network users. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy* (2010), SP '10, pp. 223–238.

[211] Wondracek, G., Holz, T., Platzer, C., Kirda, E., and Kruegel, C. Is the internet for porn? an insight into the online adult industry. In *Proceedings of the Ninth Workshop on the Economics of Information Security (WEIS)* (2010).

[212] Wright, A. Ready for a Web OS? *Communications of the ACM 52*, 12 (Dec. 2009), 16–17.

[213] Wright, C. V., Ballard, L., Coull, S. E., Monrose, F., and Masson, G. M. Spot me if you can: Uncovering spoken phrases in encrypted voip conversations. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy* (2008), pp. 35–49.

[214] Wright, J. Detecting Wireless LAN MAC Address Spoofing, 2003.

[215] Xia, H., and Brustoloni, J. C. Hardening Web browsers against man-in-the-middle and eavesdropping attacks. In *Proceedings of the 14th international conference on World Wide Web (WWW '05)* (2005), pp. 489–498.

[216] Xie, Y., and Aiken, A. Static Detection of Security Vulnerabilities in Scripting Languages. In *Proceedings of the 15th USENIX Security Symposium* (2006).

[217] Apache.org. `https://blogs.apache.org/infra/entry/apache_org_04_09_2010`.

[218] XSSed | Cross Site Scripting (XSS) attacks information and archive.

[219] The Cross-site Scripting FAQ. `http://www.cgisecurity.com/xss-faq.html`.

[220] Xu, J., and Nguyen, T. Private browsing and Flash Player 10.1. `http://www.adobe.com/devnet/flashplayer/articles/privacy_mode_fp10_1.html`.

[221] Xu, W., Bhatkar, S., and Sekar, R. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *15th USENIX Security Symposium* (August 2006), pp. 121–136.

[222] Yen, T.-F., Xie, Y., Yu, F., Peng Yu, R., and Abadi, M. Host fingerprinting and tracking on the web:privacy and security implications. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)* (2012).

[223] Yue, C., and Wang, H. Characterizing insecure JavaScript practices on the web. In *Proceedings of the 18th international conference on World wide web* (New York, NY, USA, 2009), WWW '09, ACM, pp. 961–970.

[224] Yuill, J., Zappe, M., Denning, D., and Feer, F. Honeyfiles: deceptive files for intrusion detection. *Proceedings from the Fifth Annual IEEE SMC Information Assurance Workshop*, June (2004), 116–122.

[225] Zalewski, M. *The Tangled Web: A Guide to Securing Modern Web Applications*. No Starch Press, 2011.

[226] Zeller, W., and Felten, E. W. Cross-site request forgeries: Exploitation and prevention, 2008.

[227] Zetter, K. Google Hack Attack Was Ultra Sophisticated, New Details Show. `http://www.wired.com/threatlevel/2010/01/operation-aurora/`.

[228] Zhou, Y., and Evans, D. Why Aren't HTTP-only Cookies More Widely Deployed? In *Proceedings of 4th Web 2.0 Security and Privacy Workshop (W2SP)* (2010).

[229] Zone-H: Unrestricted information. `http://zone-h.org/`.

Curriculum Vitae

Nick Nikiforakis was born on November 4th, 1985, in Athens, Greece. While his official first name is “Nikolaos”, he goes by “Nick” since the latter is shorter and less formal. Nick studied Computer Science at the University of Crete in Greece, and received his Bachelor degree in 2007. He further pursued a master’s degree in Distributed and Parallel Systems at the same university, and graduated in 2009. From 2006 till 2009, in parallel with his studies, Nick was doing security-related research at the Foundation of Research & Technology.

Nick joined KU Leuven as a PhD student in September 2009, under the supervision of Prof. Wouter Joosen and Prof. Frank Piessens. Nick is interested in all sorts of practical, hands-on security with a strong, empirical streak. He visited the University of California, Santa Barbara in the summer of 2012 (hosted by Prof. Christopher Kruegel and Prof. Giovanni Vigna), where he worked on uncovering the current practices of web-fingerprinting.

Nick is married to Freya De Leeuw and they currently live, together with their parrots, in Heverlee, Belgium.

List of publications

International Conference Articles

- Nick Nikiforakis, Frank Piessens and Wouter Joosen. HeapSentry: Kernel-assisted Protection against Heap Overflows. In: *Proceedings of the 10th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 2013)*, Berlin, Germany, pp. 177–196
- Nick Nikiforakis, Steven Van Acker, Wannes Meert, Lieven Desmet, Frank Piessens and Wouter Joosen. Bitsquatting: Exploiting bit-flips for fun, or profit?. In: *Proceedings of the 22nd International World Wide Web Conference (WWW 2013)*, Rio de Janeiro, Brazil, pp. 989–998
- Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens and Giovanni Vigna. Cookieless Monster: Exploring the Ecosystem of Web-based Device Fingerprinting. In: *Proceedings of the 34th IEEE Symposium of Security and Privacy (IEEE S&P 2013)*, San Francisco, CA, USA, pp. 541–555
- Philippe De Ryck, Nick Nikiforakis, Lieven Desmet and Wouter Joosen. TabShots: Client-side detection of tabnabbing attacks. In: *Proceedings of the 8th ACM Symposium on Information, Computer and Communications Security (ASIACCS 2013)*, Hangzhou, China, pp. 447–456
- Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens and Giovanni Vigna. You Are What You Include: Large-scale Evaluation of Remote JavaScript Inclusions. In: *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS 2012)*, Raleigh, NC, USA, pp. 736–747
- Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. 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)*, Raleigh, NC, USA, pp. 748–759

- Job Noorman, Nick Nikiforakis, and Frank Piessens. There is Safety in Numbers: Preventing Control-Flow Hijacking by Duplication. In: *Proceedings of the 17th Nordic Conference on Secure IT Systems (NordSec 2012)*, Karlskrona, Sweden, pp. 105–120

- Sebastian Lekies, Nick Nikiforakis, Walter Tighzert, Frank Piessens and Martin Johns. DEMACRO: Defense against Malicious Cross-domain Requests. In: *Proceedings of the 15th International Symposium on Research In Attacks, Intrusions and Defenses (RAID 2012)*, Amsterdam, The Netherlands, pp. 254–273

- Philippe De Ryck, Nick Nikiforakis, Lieven Desmet, Frank Piessens and Wouter Joosen. Serene: Self-Reliant Client-Side Protection against Session Fixation. In: *Proceedings of the 7th International Federated Conference on Distributed Computing Techniques (DAIS 2012)*, Stockholm, Sweden, pp. 59–72

- Nick Nikiforakis, Steven Van Acker, Frank Piessens and Wouter Joosen. Exploring the Ecosystem of Referrer-Anonymizing Services. In: *Proceedings of the 12th Privacy Enhancing Technology Symposium (PETS 2012)*, Vigo, Spain, pp. 259–278

- Steven Van Acker, Nick Nikiforakis, Lieven Desmet, Wouter Joosen and Frank Piessens. FlashOver: Automated Discovery of Cross-site Scripting Vulnerabilities in Rich Internet Applications. In: *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security (ASIACCS 2012)*, Seoul, South Korea, pp. 12–13

- Francesco Gadaleta, Nick Nikiforakis, Jan Tobias Mühlberg and Wouter Joosen. HyperForce: Hypervisor-enForced Execution of Security-Critical Code. In: *Proceedings of the 27th IFIP International Information Security and Privacy Conference (IFIP SEC 2012)*, Heraklion, Crete, Greece, pp. 126–137

- John Wilander, Nick Nikiforakis, Yves Younan, Mariam Kamkar and Wouter Joosen. RIPE: Runtime Intrusion Prevention Evaluator. In: *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC 2011)*, Orlando, USA, pp. 41–50

- Francesco Gadaleta, Nick Nikiforakis, Yves Younan and Wouter Joosen. Hello rootKitty: A lightweight invariance-enforcing framework. In:

Proceedings of the 14th Information Security Conference (ISC 2011), Xi'an, China, pp. 213–228

- Nick Nikiforakis, Wannes Meert, Yves Younan, Martin Johns and Wouter Joosen. SessionShield: Lightweight Protection against Session Hijacking. In: *Proceedings of the 3rd International Symposium on Engineering Secure Software and Systems (ESSoS 2011)*, Madrid, Spain, pp. 87–100

- Steven Van Acker, Nick Nikiforakis, Pieter Philippaerts, Yves Younan and Frank Piessens. ValueGuard: Protection of native applications against data-only buffer overflows. In: *Proceedings of the Sixth International Conference on Information Systems Security (ICISS 2010)*, Gujarat, India, pp. 156–170

- Nick Nikiforakis, Yves Younan and Wouter Joosen. HProxy: Client-side detection of SSL stripping attacks. In: *Proceedings of the 7th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2010)*, Bonn, Germany, pp. 200–218

- Nikos Nikiforakis, Andreas Makridakis, Elias Athanasopoulos and Evangelos P. Markatos. 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 2007)*, Heraklion, Greece, pp. 107–117

Peer-reviewed International Workshop Articles

- Pieter Agten, Nick Nikiforakis, Raoul Strackx, Willem De Groef and Frank Piessens. Recent Developments in Low-Level Software Security. In: *Proceedings of the 6th Workshop in Information Security Theory and Practice (WISTP 2012)*, London, UK, pp. 1–16

- Nick Nikiforakis, Wouter Joosen and Martin Johns. Abusing Locality in Shared Web Hosting. In: *Proceedings of the 4th European Workshop on System Security (EuroSec 2011)*, Salzburg, Austria, pp. 2:1–2:7

- Nick Nikiforakis, Marco Balduzzi, Steven Van Acker, Wouter Joosen and Davide Balzarotti. Exposing the Lack of Privacy in File Hosting Services. In: *Proceedings of the 4th USENIX Workshop on Large-scale Exploits and Emergent Threats (LEET 2011)*, Boston, USA, pp. 1–1

- Demetris Antoniades, Michalis Polychronakis, Nick Nikiforakis, Evangelos P. Markatos and Yiannis Mitsos. Monitoring three National Research

Networks for Eight Weeks: Observations and Implications. In: *the 6th IEEE Workshop on End-to-End Monitoring Techniques and Services (E2EMon 2008)*, Salvador, Bahia, Brazil, pp. 153–156

- Nikos Nikiforakis, Demetres Antoniades, Evangelos P. Markatos, Sotiris Ioannidis, Arne Olesbo. When Appmon met Stager. In: *the 6th IEEE Workshop on End-to-End Monitoring Techniques and Services (E2EMon 2008)*, Salvador, Bahia, Brazil, pp. 157–160

Presentations at Public Events

- Nick Nikiforakis. Web Fingerprinting: How, Who, and Why? *OWASP AppSec Research*, 2013, Hamburg, Germany
- Nick Nikiforakis. (Invited Talk) Cookieless Monster: Exploring the Ecosystem of Web-based Device Fingerprinting. *2nd SysSec Workshop*, 2013, Bochum, Germany
- Nick Nikiforakis. (Invited Talk) You are what you include: Large-scale analysis of remote JavaScript inclusions. *Web Application Security Seminar Dagstuhl*, 2012, Germany
- Nick Nikiforakis, Steven Van Acker, Wouter Joosen. Rating File Hosting Services in light of potential abuse. *ICT KTN - Cyber Security Showcase*, 2012, Brussels, Belgium
- Nick Nikiforakis. (Invited Talk) Abusing locality in Shared Web Hosting. *OWASP Netherlands Chapter Meeting*, 2011, Amsterdam, The Netherlands
- Nick Nikiforakis. Abusing locality in Shared Web Hosting. *BruCON*, 2011, Brussels, Belgium
- Nick Nikiforakis. (Invited Talk) On the Privacy of File Sharing Services, *OWASP BeNeLux*, 2010, Eindhoven, The Netherlands
- Nick Nikiforakis. Breaking Web Applications in Shared Hosting environments. CONFidence 2.0, 2010, Prague, Czech Republic
- Nick Nikiforakis. Alice Shares, Even Reads: Enumerating File Hosting Services. *AthCon*, 2010, Athens, Greece
- Nick Nikiforakis. On the Privacy of file sharing services. *OWASP AppSecDev Research 2010*, Stockholm, Sweden

FACULTY OF ENGINEERING
DEPARTMENT OF COMPUTER SCIENCE
SCIENTIFIC COMPUTING GROUP
Celestijnenlaan 200A box 2402
B-3001 Heverlee
nick.nikiforakis@cs.kuleuven.be