
Dissertations, Theses, and Masters Projects

Theses, Dissertations, & Master Projects

2010

Enhancing Web Browsing Security

Chuan Yue

College of William & Mary - Arts & Sciences

Follow this and additional works at: <https://scholarworks.wm.edu/etd>



Part of the [Computer Sciences Commons](#)

Recommended Citation

Yue, Chuan, "Enhancing Web Browsing Security" (2010). *Dissertations, Theses, and Masters Projects*. Paper 1539623575.

<https://dx.doi.org/doi:10.21220/s2-jpwx-sw57>

This Dissertation is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Dissertations, Theses, and Masters Projects by an authorized administrator of W&M ScholarWorks. For more information, please contact scholarworks@wm.edu.

Enhancing Web Browsing Security

Chuan Yue

Lanzhou, Gansu Province, China

**Master of Engineering, Xidian University, 1999
Bachelor of Engineering, Xidian University, 1996**

**A Dissertation presented to the Graduate Faculty
of the College of William and Mary in Candidacy for the Degree of
Doctor of Philosophy**

Department of Computer Science

**The College of William and Mary
August, 2010**

APPROVAL PAGE

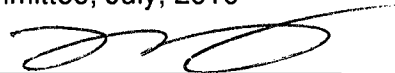
This Dissertation is submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy



Chuan Yue

Approved by the Committee, July, 2010

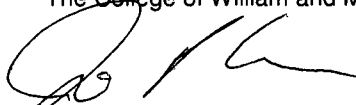


Committee Chair

Associate Professor Haining Wang, Computer Science
The College of William and Mary




Associate Professor Weizhen Mao, Computer Science
The College of William and Mary



Associate Professor Phil Kearns, Computer Science
The College of William and Mary



Associate Professor Qun Li, Computer Science
The College of William and Mary



Assistant Professor Gexin Yu, Mathematics
The College of William and Mary

ABSTRACT PAGE

Web browsing has become an integral part of our lives, and we use browsers to perform many important activities almost everyday and everywhere. However, due to the vulnerabilities in Web browsers and Web applications and also due to Web users' lack of security knowledge, browser-based attacks are rampant over the Internet and have caused substantial damage to both Web users and service providers. Enhancing Web browsing security is therefore of great need and importance.

This dissertation concentrates on enhancing the Web browsing security through exploring and experimenting with new approaches and software systems. Specifically, we have systematically studied four challenging Web browsing security problems: HTTP cookie management, phishing, insecure JavaScript practices, and browsing on untrusted public computers. We have proposed new approaches to address these problems, and built unique systems to validate our approaches.

To manage HTTP cookies, we have proposed an approach to automatically validate the usefulness of HTTP cookies at the client-side on behalf of users. By automatically removing useless cookies, our approach helps a user to strike an appropriate balance between maximizing usability and minimizing security risks. To protect against phishing attacks, we have proposed an approach to transparently feed a relatively large number of bogus credentials into a suspected phishing site. Using those bogus credentials, our approach conceals victims' real credentials and enables a legitimate website to identify stolen credentials in a timely manner. To identify insecure JavaScript practices, we have proposed an execution-based measurement approach and performed a large-scale measurement study. Our work sheds light on the insecure JavaScript practices and especially reveals the severity and nature of insecure JavaScript inclusion and dynamic generation practices on the Web. To achieve secure and convenient Web browsing on untrusted public computers, we have proposed a simple approach that enables an extended browser on a mobile device and a regular browser on a public computer to collaboratively support a Web session. A user can securely perform sensitive interactions on the mobile device and conveniently perform other browsing interactions on the public computer.

Table of Contents

Acknowledgments	xii
List of Tables	xiii
List of Figures	xv
1 Introduction	2
1.1 Motivation and Concentration	2
1.2 Contributions	4
1.2.1 Automatic HTTP Cookie Management	4
1.2.2 Transparent Phishing Protection	5
1.2.3 Characterizing Insecure JavaScript Practices on the Web	7
1.2.4 Secure and Convenient Kiosk Browsing	9
1.3 Organization	10
2 Automatic HTTP Cookie Management	12
2.1 Motivation	12
2.2 Background	15

2.3	HTTP Cookie Measurement	19
2.3.1	Website Selection and Crawling	19
2.3.2	Measurement Results	20
2.4	CookiePicker Design	25
2.4.1	Regular and Hidden Requests	26
2.4.2	Forward Cookie Usefulness Marking	28
2.4.3	Backward Error Recovery	30
2.5	HTML Page Difference Detection	32
2.5.1	Restricted Simple Tree Matching	32
2.5.1.1	Tree Edit Distance	33
2.5.1.2	Top-Down Distance	34
2.5.1.3	Restricted Simple Tree Matching	36
2.5.1.4	Normalized Top-Down Distance Metric	38
2.5.2	Context-aware Visual Content Extraction	38
2.5.3	Making Decision	41
2.6	System Evaluation	42
2.6.1	Implementation	42
2.6.2	Evaluation	44
2.6.2.1	First Set of Experiments	45
2.6.2.2	Second Set of Experiments	47
2.7	Discussions	49
2.7.1	Possible Evasion Techniques	49
2.7.2	Evasion Sources	50

2.7.3	Concerns about Using CookiePicker	51
2.8	Related Work	52
2.9	Summary	55
3	Transparent Phishing Protection	56
3.1	Motivation	56
3.2	Background	61
3.3	Design	64
3.3.1	Design Overview	64
3.3.1.1	How It Works	65
3.3.1.2	Design Assumption	66
3.3.1.3	Design Objectives	67
3.3.2	Offensive Line	67
3.3.2.1	Massiveness	68
3.3.2.2	Indiscernibility	71
3.3.2.3	Usability	73
3.3.3	Defensive Line	74
3.3.3.1	Working Mechanism	74
3.3.3.2	Deployment of Defensive Line	80
3.3.3.3	Scale-Independency Properties	81
3.4	Implementation	81
3.4.1	Information Extraction Module	82
3.4.2	Bogus Credential Generation Module	83

3.4.3	Request Submission Module	84
3.4.4	Response Process Module	87
3.5	Evaluation	89
3.5.1	Testbed Experiments	89
3.5.2	Phishing Site Experiments	89
3.5.3	Legitimate Site Experiments	91
3.6	Discussions	92
3.6.1	Deployment Scale	93
3.6.2	Massive Deployment Preparation	93
3.6.3	Limitations of BogusBiter	94
3.6.3.1	Offline Evasions	95
3.6.3.2	Online Evasions	96
3.7	Related Work	100
3.8	Summary	102
4	Characterizing Insecure JavaScript Practices on the Web	103
4.1	Motivation	103
4.2	Background	107
4.3	Methodology	109
4.3.1	Instrumentation Framework	109
4.3.2	Analysis Toolkit	112
4.4	Data Set	115
4.5	Results and Analysis	116

4.5.1	Overall JavaScript Presence	116
4.5.2	Insecure JavaScript Inclusion	118
4.5.2.1	Results and Analysis	118
4.5.2.2	Safe Alternatives to Insecure Inclusion	122
4.5.3	Insecure JavaScript Dynamic Generation	122
4.5.3.1	DJS Presence by Category and TLD	123
4.5.3.2	DJS Instance Summary	124
4.5.3.3	Structural Analysis of eval-generated DJS	126
4.5.3.4	Safe Alternatives to eval()	129
4.5.3.5	Structural Analysis of Other Types of DJS	130
4.5.3.6	Safe Alternatives to jscode Generation via document.write() and innerHTML	131
4.5.4	Event Handler Registration	132
4.6	Related Work	133
4.7	Summary	133
5	Secure and Convenient Kiosk Browsing	135
5.1	Motivation	135
5.2	Related Work	139
5.2.1	Securing Application or Data Access	139
5.2.2	Securing User Authentication or Input	140
5.2.3	Verifying Software Integrity	140
5.2.4	Securing Web Browsing Sessions	141

5.3	Design	142
5.3.1	A Motivating Example	142
5.3.2	Threat Model and Assumptions	144
5.3.3	Architecture Design	146
5.3.3.1	Connection Control	148
5.3.3.2	Request Authentication	149
5.3.3.3	Request Processing	150
5.3.3.4	Response Generation	151
5.3.3.5	Initial Webpage	156
5.4	Implementation	156
5.5	Security Analysis	158
5.6	Evaluation	161
5.6.1	Usability Evaluation	161
5.6.1.1	Participants	161
5.6.1.2	Scenario and Procedure	161
5.6.1.3	Data Collection	164
5.6.1.4	Results and Analysis	164
5.6.2	Performance and Feasibility Evaluation	166
5.7	Summary	168
6	Conclusions and Future Work	170
6.1	Conclusions	170
6.2	Future Work	171

Bibliography 173

Vita 186

Dedicated to my parents, my wife and daughter, and my whole family
for their consistent love and support.

0

ACKNOWLEDGMENTS

This long journey to a Ph.D. in computer science could not be accomplished without the support and help from many people. First and foremost I want to thank my advisor Professor Haining Wang. It has been my great honor and pleasure to be his Ph.D. student for the past five years. I sincerely thank him for all his guidance, support, encouragement, trust, time, and effort that made this dissertation possible. In addition to appreciating his many qualities as an excellent advisor, I especially want to thank him for always giving me the freedom to explore on my own, always setting a high standard for my research, always being available to meet and discuss projects with me, as well as always treating me and other students with respect.

I am very grateful to other committee members Professor Weizhen Mao, Professor Phil Kearns, Professor Qun Li, Professor Gexin Yu, and my former committee member Professor Evgenia Smirni, for their valuable feedback and suggestions that helped me improve the quality of this dissertation. Particularly, I am heartily thankful to Professor Smirni, Professor Mao, Professor Kearns, and my advisor for their tremendous support in helping me get a job. I am also thankful to Professor Andreas Stathopoulos, Professor Xiaodong Zhang, Professor Bruce Lowekamp, Professor Dimitris Nikolopoulos, Professor Barry Lawson, and Dr. Richard Tran Mills for their help in my first-year Ph.D. study.

My sincere appreciation also goes to graduate center course instructors Ms. Barbara Monteith and Ms. Robin Cantor-Cooke. Barbara has helped me a lot in improving my academic writing skills, and Robin has helped me a lot in improving my academic presentation skills. Barbara and Robin have also given me great help in my job application process. I also want to sincerely thank administrative members Ms. Vanessa Godwin and Ms. Jacquelyn Johnson, as well as all the techies in our computer science department. Their professional and efficient help made my Ph.D. study and research a pleasant experience.

I also thank many former and current students for their help, collaboration, and friendship. Qi Zhang gave me suggestions on applying for my first-year Ph.D. assistantship. Mengjun Xie helped me in collecting and processing trace for the CookiePicker project. Zi Zhu helped me in testing the RCB framework. Eli Courtwright and Benjamin Strahs generously invited me to be deeply involved in their research projects. Zhenyu Wu provided a great deal of useful information to me with his extensive software and hardware experience. Thanks also go to Heng Yin, Chiu Chiang Tan, Bo Sheng, Ningfang Mi, Haodong Wang, Steven Gianvecchio, Meghan Reville, Ruth Lamprecht, Zhen Ren, Duy Le, Feng Mao, Kai Tian, Yunlian Jiang, Zheng Zhang, Yixun Liu, and many others. It has been a very nice experience to learn from all of them.

Finally, I owe my deepest gratitude to my parents and parents-in-law, my wife Weiyang and daughter Catherine, my brothers and sisters-in-law, my nieces, my brother-in-law and his family, as well as many other relatives and friends for their consistent love, help, and support. My life is meaningful because of them.

List of Tables

2.1	Statistics of the selected websites and their usage of HTTP cookies.	21
2.2	Online testing results for thirty websites (S1 to S30).	46
2.3	Online testing results for 6 websites (P1 to P6) that have useful persistent cookies.	49
3.1	Substitution from the original username/password pair (<i>mcsmith/Fuzzycat15</i>).	77
3.2	Derivation from the username/password pair (<i>lcsmith/Fuzzycat05</i>).	77
3.3	The 20 legitimate websites.	92
4.1	Category breakdown by top-level domain.	115
4.2	JavaScript presence by category and top-level domain.	117
4.3	DJS presence by category and top-level domain.	123
4.4	DJS instance summary for pre-onload/post-onload phases.	125
4.5	The 17 categories of eval-generated DJS instances.	128
4.6	Structural analysis of DJS instances generated by the <code>document.write()</code> method, <code>innerHTML</code> property, and DOM methods.	130
5.1	The 10 tasks in procedure A.	162

5.2 The 18 tasks in procedure B. 163

5.3 The 16 close-ended questions. 165

5.4 Page size and response generation time of five homepages. 167

List of Figures

2.1	Internet Explorer 7 advanced privacy settings.	18
2.2	CDFs of the number of all cookies, the number of session cookies, the number of persistent cookies per website, respectively.	22
2.3	CDFs of cookie size in bytes for all cookies, session cookies, and persistent cookies, respectively.	23
2.4	CDFs of cookie PATH depth for all cookies, session cookies, and persistent cookies, respectively.	24
2.5	CDF of persistent cookie lifetime.	25
2.6	HTTP requests/responses in a single webpage view.	26
2.7	Data flow inside Gecko.	27
2.8	The five-step FORCUM process.	29
2.9	The restricted simple tree matching algorithm.	35
2.10	(a) Tree A, (b) Tree B.	36
2.11	The text content extraction algorithm.	40
2.12	CookiePicker decision algorithm.	42
2.13	CookiePicker user interface.	43

3.1	(a) A phishing site designed to attack eBay users, (b) Firefox 2 phishing warning mechanism.	61
3.2	Anti-phishing with BogusBiter.	65
3.3	Expected number of tries for a phisher to single out: (a) one real credential, (b) all real credentials.	71
3.4	The Stolen Credential Identification (SCI) procedure.	78
3.5	Implementation of BogusBiter as a Firefox 2 browser extension.	82
3.6	Delay caused by BogusBiter on: (a) phishing sites, (b) legitimate sites, under different set size S	91
4.1	Overview of the instrumentation framework and analysis toolkit.	110
4.2	High-level AST signature creation and matching procedure.	114
4.3	Cumulative distribution of the 4,517 JavaScript file inclusion domains in terms of their outdegree values.	119
4.4	Cumulative distribution of the 1,985 JavaScript file hosting domains in terms of their indegree values.	120
4.5	Cumulative distribution of the webpages in terms of IPP (Instance Per Page) for (a) eval-generated, (b) write-generated, (c) innerHTML-generated, and (d) DOM-generated DJS instances.	126
4.6	Cumulative distribution of the AST trees in terms of the height of an AST tree.	127
5.1	Kiosk browsing environment.	143
5.2	SessionMagnifier high-level architecture.	146

5.3	Response generation procedure.	152
5.4	Performing the task TB9 on Fennec.	164
5.5	Mean ratings to questions Q1 to Q6.	166

Enhancing Web Browsing Security

Chapter 1

Introduction

Web-centric computing is a significant trend in computing, and Web browsing has become an integral part of our lives. People use browsers to perform many important activities such as banking, shopping, and bill-paying; they also use browsers to search, communicate, and collaborate, as well as to teach, learn, and entertain. Meanwhile, Web browsers have evolved dramatically since the invention of the first Web client in 1990 [186]. Web browsers are no longer simple HTML document renderers; they have increasingly become the dominant platform for both current and future end-user applications. This fascinating trend manifests the need and importance of providing modern Web browsers with higher security assurance and better functionality.

1.1 Motivation and Concentration

Traditionally, attackers mainly focused on directly breaking into Web servers to acquire sensitive information or damage the systems. However, in recent years, Web browsers have

gained increasing popularity as new attack vectors for launching various severe attacks such as drive-by download, cross-site scripting, phishing, privacy, and even large-scale denial of service attacks. Browser-based attacks are rampant over the Internet and have caused substantial damage to both Web users and service providers. Meanwhile, Internet security threat trends [159] show that malicious activities have increasingly become Web-based, attackers are more and more targeting end users, the online underground economy has matured, and attackers are able to rapidly adapt their attacking techniques. Enhancing Web browsing security is therefore of great need and importance, and it has also become much more challenging than before.

This dissertation concentrates on enhancing the Web browsing security through exploring and experimenting with new approaches and software systems. Specifically, we have systematically studied four challenging Web browsing security problems: HTTP cookie management, phishing, insecure JavaScript practices, and browsing on untrusted public computers. These problems are real-world important problems that are closely related to a few top cyber security threats. For example, all the four problems are tightly related to the threat of online identity theft, which is the greatest fear for Internet users [159, 174]. For another example, the problem of insecure JavaScript practices is also tightly related to cross-site scripting, which is the number one Web application vulnerability of all time [33, 190, 193].

In studying these four Web browsing security problems, we have carefully surveyed state-of-art solutions in literature and also carefully experimented with state-of-practice security features in modern Web browsers. We have made important observations and proposed novel approaches to address these problems. We have also built unique browser-based software systems to validate our approaches. In the next section, we briefly introduce

these four Web browsing security problems and our contributions. We detail the motivation, design, implementation, and evaluation of each work in the following four Chapters 2, 3, 4, and 5, respectively.

1.2 Contributions

The contributions of this dissertation include an automatic HTTP cookie management system, a transparent phishing protection system, a large-scale measurement of insecure JavaScript practices on the Web, and a simple approach to secure and convenient kiosk browsing.

1.2.1 Automatic HTTP Cookie Management

HTTP cookies are designed and widely used to provide many beneficial features such as maintaining session state and enabling webpage personalization. However, despite their importance and usefulness, HTTP cookies have raised significant user privacy and security concerns. This is because HTTP cookies can be used to monitor users' browsing habits, and possibly to associate what you have looked at with who you are. Moreover, stolen cookies may also incur severe security problems; for example, they can be used to impersonate other users and to gain access to session-protected services and resources.

Through a systematic study of HTTP cookie management on modern Web browsers and a large-scale measurement of cookie usage on over five thousand websites, we made the observation that handling first-party persistent cookies is the most challenging problem in HTTP cookie management. First-party persistent cookies are stored on the hard disk of a user's computer until they expire or are deleted by a user. Websites often set a long lifetime

for first-party persistent cookies, thus leaving plenty of room for attackers to manipulate or steal those cookies. Unfortunately, modern Web browsers only provide very limited cookie management support, and often require users to manually define policies.

To address this problem, we designed *CookiePicker* [126, 127], a system that can automatically verify the usefulness of first-party persistent cookies at the client-side. *CookiePicker* identifies those cookies that can cause perceivable changes on a webpage as useful, while simply classifying the rest as useless. Useless cookies will be finally removed from the browser to reduce potential security and privacy risks. *CookiePicker* therefore helps users achieve the maximum benefit brought by cookies, while minimizing the potential risks. We implemented *CookiePicker* as an extension to the Firefox Web browser, and evaluated its efficacy and overhead through live experiments. The experimental results indicate that *CookiePicker* can automatically make high-accuracy cookie usage decisions with very low running overhead. We detail this work in Chapter 2.

1.2.2 Transparent Phishing Protection

Phishing is one of the most common Internet frauds, and it has caused serious damages to Internet users ranging from denied account access to substantial financial loss. While it seems that phishing is so simple, it is a very challenging problem because it is a third-wave semantic attack that targets the way humans assign meaning to content [182]. It is also important to note that phishing is not far away from us. For example, according to the IT Administration of our university, phishing targets William and Mary online accounts as well. Individuals on our campus frequently became the victims of phishing attacks, and serious consequences have occurred. Many anti-phishing mechanisms focus on helping

users verify whether a website is genuine. They endeavor to prevent users from being tricked into revealing their credentials to phishing sites. Nevertheless, prevention-based approaches alone are insufficient to shield vulnerable users from “biting the bait” and defeat phishers because human users are the weakest link in the security chain. The ever-increasing prevalence and severity of phishing attacks clearly indicate that anti-phishing is still a daunting challenge.

Two observations motivated us to explore a quite different approach. First, currently, it is trivial for phishers to verify their collected credentials. The majority of credentials collected by a phishing site are victims’ real credentials. Meanwhile, when phishers go to a legitimate site to verify their collected credentials, it is very hard for the legitimate site to differentiate phishers’ credential verification activities from users’ regular login activities. Therefore, we had such a question: can we make it harder for phishers to verify their collected credentials? Second, browsers warn instead of block users’ connections to suspected phishing sites. Although remarkable advances in automatic phishing detection have been achieved, false positives may still happen. Therefore, giving warnings and expecting users to leave suspected phishing sites have become the most common actions taken by modern browsers. However, usability studies have demonstrated that many users still ignore the strong phishing warnings. Therefore, we had another question: can we leverage the power of automatic phishing detection and can we effectively transform this power into the power of automatic fraud protection?

With these observations and questions, we explored a quite different approach to protect against phishing attacks with “bogus bites.” We developed *BogusBiter* [122, 125], a unique client-side anti-phishing tool, which transparently feeds a relatively large number of bogus

credentials into a suspected phishing site. BogusBiter conceals a victim's real credential among bogus credentials, and moreover, it enables a legitimate website to identify stolen credentials in a timely manner. BogusBiter leverages the power of client-side automatic phishing detection techniques, and it is complementary to existing preventive anti-phishing approaches. Seamlessly integrated with the phishing detection and warning mechanisms in modern Web browsers, BogusBiter is transparent to users. We implemented BogusBiter as an extension to the Firefox Web browser, and evaluated its efficacy through real experiments on both phishing and legitimate websites. Our experimental results indicate that it is promising to use BogusBiter to transparently protect against phishing attacks. We detail this work in Chapter 3.

1.2.3 Characterizing Insecure JavaScript Practices on the Web

JavaScript is an interpreted programming language widely used for client-side scripting. JavaScript code embedded or included in webpages runs locally in a user's Web browser, and it is mainly used by websites to enhance the interactivity and functionality of their webpages. However, because JavaScript is equipped with a powerful and diverse set of capabilities in modern Web browsers, it has also become the weapon of choice for attackers. Many severe attacks such as drive-by-download, cross-site scripting, as well as the aforementioned cookie stealing and phishing, are closely related to JavaScript. To a very large extent, JavaScript has become a central battle ground of the Web security.

A great deal of attention has been paid to the JavaScript-related vulnerabilities that could directly lead to security breaches. However, little attention has been given to the insecure practices of using JavaScript on legitimate websites. Similar to other insecure practices

on legitimate websites such as using customers' social security numbers as their login IDs, insecure JavaScript practices may not necessarily result in direct security breaches, but they could definitely cultivate the creation of new attack vectors and greatly increase the risks of browser-based attacks.

This observation motivated us to identify insecure JavaScript practices on legitimate websites, to analyze their severity, and to highlight their risks. To achieve this goal, we devised an execution-based measurement approach [123]. The key idea of this approach is to first use an instrumented Web browser to non-intrusively obtain JavaScript execution information on different websites, and then use an offline analysis toolkit to sufficiently analyze the collected trace information. Using our instrumentation framework and analysis toolkit, we performed the first large-scale measurement study of the insecure JavaScript practices on the Web. Our focus is on the insecure practices of JavaScript inclusion and dynamic generation, and we examined their severity and nature on 6,805 unique websites. Our measurement results reveal that insecure JavaScript practices are common at various websites: (1) at least 66.4% of the measured websites manifest the insecure practices of including JavaScript files from external domains into the top-level documents of their webpages; (2) over 44.4% of the measured websites use the dangerous `eval()` function to dynamically generate and execute JavaScript code on their webpages; and (3) in JavaScript dynamic generation, using the `document.write()` method and the `innerHTML` property is much more popular than using the relatively secure technique of creating script elements via DOM (Document Object Model) [191] methods. Our in-depth analysis indicates that safe alternatives to those insecure JavaScript practices exist in common cases and ought to be adopted by website developers and administrators for reducing potential security risks.

We detail this work in Chapter 4.

1.2.4 Secure and Convenient Kiosk Browsing

Many kiosk environments such as airport lounges and hotel business centers provide people with Internet-connected public computers to facilitate ubiquitous Web access. Those public computers often have high-speed network connections. They are also convenient to use because they normally have full-size keyboards and large displays. People who do not own a computer or carry a laptop with them frequently use those public computers to browse the Web. Unfortunately, public computers are usually far less trustworthy than peoples' own computers. Public computers are used by many people to run different applications and visit various websites; therefore, it is very likely for them to be infected with malware. Simply searching "public computer security" online, we can find numerous articles suggesting that people should not use public computers to perform sensitive activities.

To secure kiosk computing environments, researchers have proposed a number of solutions. Many of those solutions focus on specific objectives such as securing application and data access, securing user authentication and input, or verifying software integrity. However, those solutions cannot be easily adopted to secure an entire kiosk browsing session. A few solutions do have the objective of securing an entire kiosk browsing session, but they are very complex, not very secure, and not practical. They are very complex and not very secure because they all use the browser on the untrusted public computer to directly access remote Web servers. They are not practical because they often require support from extra proxies or extra authentication and key-exchange processes.

We proposed *SessionMagnifier* [124], a simple approach to secure and convenient kiosk

browsing. The key idea of SessionMagnifier is to enable an extended browser on a mobile device and a regular browser on a public computer to collaboratively support a Web session. This approach simply requires a SessionMagnifier browser extension to be installed on a trusted mobile device. No third-party proxy is needed, no Web server modification is needed, and no installation or configuration on the untrusted public computer is needed. A user can securely perform sensitive interactions on the mobile device and conveniently perform other browsing interactions on the public computer. We implemented SessionMagnifier for Mozilla's Fennec mobile browser and evaluated it on a Nokia N810 Internet Tablet. The evaluation and analysis demonstrate that SessionMagnifier is simple, secure, and usable. We detail this work in Chapter 5. During the process of developing SessionMagnifier, we realized that the technique we invented to solve this specific kiosk browsing problem could be extended to build a simple and practical framework for real-time collaborative browsing. Therefore, we made the decision to immediately build our RCB (Real-time Collaborative Browsing) framework [121]. I do not include the RCB work [121] in this dissertation just because it is not directly related to enhancing Web browsing security.

1.3 Organization

The remainder of this dissertation is organized as follows. Chapter 2 presents the details of our work on automatic HTTP cookie management. We describe a measurement study of HTTP cookie usage on the Web, and then fully present our CookiePicker system. Chapter 3 presents the details of our work on transparent phishing protection. We present the design and implementation of our BogusBiter system, and we also discuss its deployment and lim-

itations. Chapter 4 presents the details of our work on characterizing insecure JavaScript practices on the Web. We introduce our JavaScript measurement framework and analysis toolkit, and then completely present and analyze a large-scale measurement study of insecure JavaScript inclusion and dynamic generation practices on the Web. Chapter 5 presents the details of our work on secure and convenient kiosk browsing. We present our SessionMagnifier system and analyze its security and usability advantages in supporting kiosk browsing. Finally, Chapter 6 concludes the dissertation and presents future work.

Chapter 2

Automatic HTTP Cookie Management

In this chapter, we present our work on automatic HTTP cookie management. We advocate that useless first-party persistent cookies should be removed from a user's browser, so that potential security and privacy risks can be reduced. We present a pure client-side solution that can automatically verify the usefulness of first-party persistent cookies on behalf of a user. By removing those useless cookies, our solution can help a user achieve the maximum benefit brought by HTTP cookies while minimizing the potential security and privacy risks.

2.1 Motivation

HTTP Cookies, also known as Web cookies or just cookies, are small parcels of text sent by a server to a Web browser and then sent back unchanged by the browser if it accesses that

server again [155]. Cookies are originally designed to carry information between servers and browsers so that a stateful session can be maintained within the stateless HTTP protocol. For example, online shopping websites use cookies to keep track of a user's shopping basket. Cookies make Web applications much easier to write, and thereby have gained a wide range of usage since debut in 1995. In addition to maintaining session states, cookies have also been widely used for personalizing, authenticating, and tracking user behaviors.

Despite their importance and usefulness, cookies have been of major concern for privacy. As pointed out by Kristol in [63], the ability to monitor browsing habits, and possibly to associate what you have looked at with who you are, is the heart of the privacy concern that cookies raise. For example, a lawsuit alleged that DoubleClick Inc. used cookies to collect Web users' personal information without their consent [14]. Moreover, vulnerabilities of Web applications or Web browsers can be exploited by attackers to steal cookies directly, leading to severe security and privacy problems [34, 47, 106, 107].

As the general public has become more aware of cookie privacy issues, a few privacy options have been introduced into Web browsers to allow users to define detailed policies for cookie usage either before or during visiting a website. However, these privacy options are far from enough for users to fully utilize the convenience brought by cookies while limiting the possible privacy and security risks. What makes it even worse is that most users do not have a good understanding of cookies and often misuse or ignore these privacy options [39].

Using cookies can be both beneficial and harmful. The ideal cookie-usage decision for a user is to enable and store useful cookies, but disable and delete harmful cookies. It has long been a challenge to design effective cookie management schemes that can help users make the ideal cookie-usage decision. On one hand, determining whether some cookies

are harmful is almost impossible, because very few websites inform users how they use cookies. Platform for Privacy Preferences Project (P3P) [177] enables websites to express their privacy practices but its usage is too low to be a practical solution. On the other hand, determining whether some cookies are useful is possible, because a user can perceive inconvenience or webpage differences if some useful cookies are disabled. For instance, if some cookies are disabled, online shopping may be blocked or preference setting cannot take into effect. However, current Web browsers only provide a method, which asks questions and prompts options to users, for making decision on each incoming cookie. Such a method is costly [74] and very inconvenient to users.

In this work, we first conduct a large-scale cookie measurement for investigating the current cookie usage on various websites. Our major measurement findings show that the pervasive usage of persistent cookies and their very long lifetimes clearly highlight the demand for removing useless persistent cookies to reduce the potential privacy and security risks. Then we present *CookiePicker*, a system that automatically makes cookie usage decisions on behalf of a Web user. *CookiePicker* enhances the cookie management for a website by using two processes: a training process to mark cookie usefulness and a tuning process to recover possible errors. *CookiePicker* uses two complementary algorithms to effectively detect HTML page difference online, and we believe that these two algorithms have the potential to be used by other online tools and applications. Based on the two HTML page difference detection algorithms, *CookiePicker* identifies those cookies that cause perceivable changes on a webpage as useful, while simply classifying the rest as useless. Subsequently, *CookiePicker* enables useful cookies but disables useless cookies. All the tasks are performed without user involvement or even notice.

Although it is debatable whether defining useful cookies as those that lead to perceivable changes in webpages retrieved is the best choice, so far this definition is the most reasonable measure at the browser side and it is also used in [97]. The reasons mainly lie in that very few websites tell users the intention of their cookie usage, P3P usage is still very low, and many websites use cookies indiscriminately [9].

We implemented CookiePicker as a Firefox Web browser extension, and validated its efficacy through live experiments over a variety of websites. Our experimental results demonstrate the distinct features of CookiePicker, including (1) fully automatic decision making, (2) high accuracy on decision making, and (3) very low running overhead.

The remainder of this chapter is structured as follows. Section 2.2 presents the background of cookies. Section 2.3 shows our cookie measurement results. Section 2.4 describes the design of CookiePicker. Section 2.5 details the two HTML page difference detection algorithms used by CookiePicker. Section 2.6 presents the implementation of CookiePicker and its performance evaluation. Section 2.7 discusses the potential evasions against CookiePicker as well as some concerns about using CookiePicker. Section 2.8 reviews related work, and finally, Section 2.9 summarizes this work.

2.2 Background

HTTP cookies allow an HTTP-based service to create stateful sessions that persist across multiple HTTP transactions [76]. When a server receives an HTTP request from a client, the server may include one or more `Set-Cookie` headers in its response to the client. The client interprets the `Set-Cookie` response headers and accepts those cookies that do not

violate its privacy and security rules. Later on, when the client sends a new request to the original server, it will use the `Cookie` header to carry the cookies with the request [61].

In the `Set-Cookie` response header, each cookie begins with a `NAME=VALUE` pair, followed by zero or more semi-colon-separated attribute-value pairs. The `NAME=VALUE` pair contains the state information that a server attempts to store at the client side. The optional attributes `Domain` and `Path` specify the destination domain and the targeted URL path for a cookie. The optional attribute `Max-Age` determines the lifetime of a cookie and a client should discard the cookie after its lifetime expires.

In general, there are two different ways to classify cookies. Based on the origin and destination, cookies can be classified into first-party cookies, which are created by the website we are currently visiting; and third-party cookies, which are created by a website other than the one we are currently visiting. Based on lifetime, cookies can be classified into session cookies, which have zero lifetime and are stored in memory and deleted after the close of the Web browser; and persistent cookies, which have non-zero lifetime and are stored on a hard disk until they expire or are deleted by a user. A recent extensive investigation of the use of first-party, third-party, session, and persistent cookies was carried out by Tappenden and Miller [103].

Third-party cookies bring almost no benefit to Web users and have long been recognized as a major threat to user privacy since 1996 [63]. Therefore, almost all the popular Web browsers, such as Microsoft Internet Explorer and Mozilla Firefox, provide users with the privacy options to disable third-party cookies. Although disabling third-party cookies is a very good start to address privacy concerns, it only limits the profiling of users from third-party cookies [63], but cannot prevent the profiling of users from first-party cookies.

First-party cookies can be either session cookies or persistent cookies. First-party session cookies are widely used for maintaining session states, and pose relatively low privacy or security threats to users due to their short lifetime. Therefore, it is quite reasonable for a user to enable first-party session cookies.

First-party persistent cookies, however, are double-edged swords. As we will show in Section 2.3, first-party persistent cookies could stay on a user's disk for a few years if not been removed. Some cookies perform useful roles such as setting preferences. Some cookies, however, provide no benefit but pose serious privacy and security risks to users. For instance, first-party persistent cookies can be used to track the user activity over time by the original website, proxies, or even third-party services. Moreover, first-party persistent cookies could be stolen or manipulated by three kinds of long-standing attacks: (1) cross-site scripting (XSS) attacks that exploit Web applications' vulnerabilities [58, 106, 193], (2) attacks that exploit Web browser vulnerabilities [15, 88, 107], and (3) attacks that could directly steal and control cookies launched by various malware and browser hijackers such as malicious browser extensions [57, 67, 94]. These attacks can easily bypass the *same origin policy* [180] of modern Web browsers, which protects cookies stored by one origin from accessing by a different origin. As an example, recently cookie-stolen related XSS vulnerabilities were even found in Google's hosting services [195, 197]; and the flaws in Internet Explorer 7 and Firefox could enable attackers to steal and manipulate cookies stored on a PC's hard drive [196].

Disabling third-party cookies (both session and persistent) and enabling first-party session cookies have been supported by most Web browsers. The hardest problem in cookie management is how to handle first-party persistent cookies. Currently Web browsers only

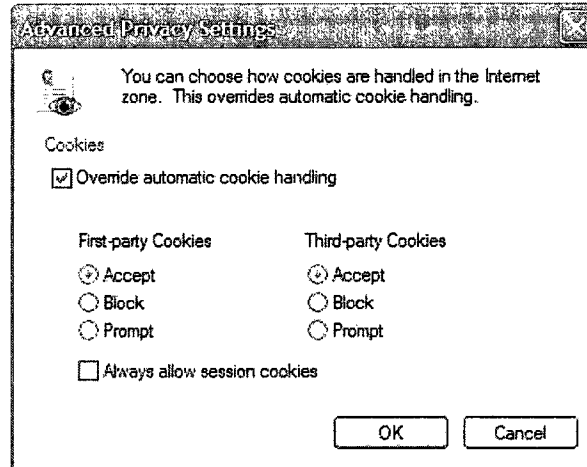


Figure 2.1: Internet Explorer 7 advanced privacy settings.

have very limited functions such as automatically accepting or blocking all first-party persistent cookies, or prompting each of them to let users manually make decisions. Figure 2.1 shows the advanced privacy settings of Internet Explorer 7, in which the functions provided to control first-party persistent cookies are very cumbersome and impractical to use. Therefore, the focus of this work is on first-party persistent cookies, that is, how to automatically manage the usage of first-party persistent cookies on behalf of a user¹. Instead of directly addressing XSS and various Web browser vulnerabilities, CookiePicker reduces cookie privacy and security risks by removing useless first-party cookies from a user's hard disk. Here we assume that the hosting website is legitimate, since it is worthless to protect the cookies of a malicious site.

¹The design of CookiePicker and its decision algorithms can be easily applied to other kinds of cookies as well if needed.

2.3 HTTP Cookie Measurement

To comprehensively understand the usage of HTTP cookies over the Internet, we conducted a large-scale measurement study in December 2009. We chose over five thousand websites from *directory.google.com*, an online website directory containing millions of websites categorized by topic. Then, we used a customized webpage retrieval tool `wget` [150] to identify those websites that set cookies at client side. We only consider first-party cookies, including both session cookies and persistent cookies, in this measurement study.

2.3.1 Website Selection and Crawling

Since there are numerous websites on the Internet, we select the websites for measurement study based on the following two requirements: diversity and representativeness. As with [79], the selection pool we use is *directory.google.com*, in which websites are organized into hierarchical categories and listed in Google page-rank order indicating the relevance to the classified category. There are fifteen top-level categories in *directory.google.com*. Our selection covers thirteen of them, except categories “world” and “regional”. These two are avoided due to the existence of many non-English websites. Since a top-level category has multiple sub-categories which also have further sub-categories, each top-level category may consist of hundreds of thousands of websites. Thus, we only choose the websites listed in the top-level categories and their immediate sub-categories. To select representative websites, we filter out those websites with page-rank less than 0.25². There may exist duplicate websites inside a category or between two categories. We first remove the duplicate

²Page-rank value is between 0 and 1. The bigger, the more relevant.

intra-category websites and then move all the duplicate inter-category websites into a new category—“Multi-category”. After filtering and removing, the total number of unique websites chosen for our study is 5,393. The number of websites in each category is listed in Table 2.1, where the second column lists the total number of websites in the corresponding category and the third column shows the number and percentage of the websites that set cookies at client side. The fourth, fifth, and sixth columns of Table 2.1 show the number of the websites that set only session cookies, only persistent cookies, and both session and persistent cookies, respectively.

We instruct the customized `wget` to crawl the selected websites and save the corresponding cookies carried in the HTTP response headers. To simulate a user surfing the Web, we turn on the recursive option in `wget` so that `wget` can recursively retrieve webpages. The maximum depth of recursion level is set to two. We instruct `wget` to only save first-party cookies. To avoid crawling being blocked or running too long, we use a random wait time varied from zero to two seconds between consecutive retrievals, and limit the crawling time on a single website within six minutes.

2.3.2 Measurement Results

After Web crawling, we found that 2,341 (43%) websites in total have set at least one first-party cookie at client side. The average number of retrieved unique URLs per website for these 2,341 websites is 538. The percentage of websites setting cookies in each category varies from 33% to 63%, as shown in Table 2.1. These numbers are conservative since `wget` cannot obtain the cookies set by the webpages that require a user login or carried by the client-side JavaScript. Even so, the results clearly show the pervasive cookie usage among

Category	Web Sites	Cookies	Session Only	Persistent Only	Both
Arts	342	113 (33%)	55	21	37
Business	602	265 (44%)	181	28	56
Computers	512	190 (37%)	108	33	49
Games	151	51 (34%)	19	12	20
Health	303	148 (49%)	82	17	49
Home	324	157 (48%)	83	23	51
News	483	176 (36%)	86	51	39
Recreation	351	173 (49%)	86	36	51
Reference	308	134 (44%)	69	17	48
Science	537	215 (40%)	118	48	49
Shopping	461	290 (63%)	93	67	130
Society	214	78 (36%)	35	17	26
Sports	673	287 (43%)	152	51	84
Multi-category	132	64 (48%)	23	24	17
Total	5,393	2,341 (43%)	1,190	445	706

Table 2.1: Statistics of the selected websites and their usage of HTTP cookies.

various types of websites.

For those websites that use cookies, we compute the CDFs (Cumulative Distribution Functions) of the number of cookies, the number of session cookies, and the number of persistent cookies per website, respectively, and draw them in Figure 2.2. Note that the X-axis is in logarithmic scale. We are interested in these numbers because Web browsers usually set limits on the total number of cookies and the number of cookies per domain, in order to save memory and disk space on a user's computer. For example, the maximum default number of cookies per domain and maximum number of total cookies are set to 50 and 1,000 respectively in Firefox. From Figure 2.2, we can see that about 99% of the websites set less than 10 cookies at client side and none of them sets more than 40 cookies. This indicates that all websites under study follow the basic rule that a website should not set too many cookies to a client.

Web browsers usually also set limit on the size of a cookie, which is mainly determined by the length of cookie value. For example, Firefox limits the length of a cookie NAME/VALUE

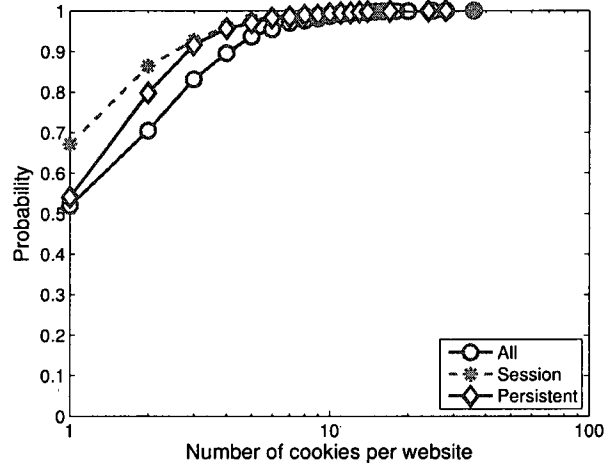


Figure 2.2: CDFs of the number of all cookies, the number of session cookies, the number of persistent cookies per website, respectively.

pair up to 4,096 bytes. Figure 2.3 depicts the CDFs of cookie size in bytes for all cookies, session cookies, and persistent cookies, respectively. Around 94% of cookies are less than 100 bytes and none of them is greater than 4,096 bytes.

If cookie is enabled during browsing a website, the path of URL will determine what cookies to be transmitted together with the HTTP request to server. If the `Path` attribute of a cookie matches the prefix of the URL path, the cookie previously set will be sent back. We examine the `Path` depth for cookies, session cookies, and persistent cookies, respectively, and draw their CDFs in Figure 2.4. Surprisingly, near 90% of cookies have root “/” as their `Path` attribute values, which implies that these cookies will be sent back to the original websites for any URL requests to those sites. Although some cookies may need to be sent back to the Web servers for any URL on the servers, the abnormally high percentage suggests that the `Path` attributes of many cookies may not be appropriately set. As a consequence of this inappropriate setting, many requests carry cookies that are functionally unnecessary: the cookies neither affect responses nor make difference to server

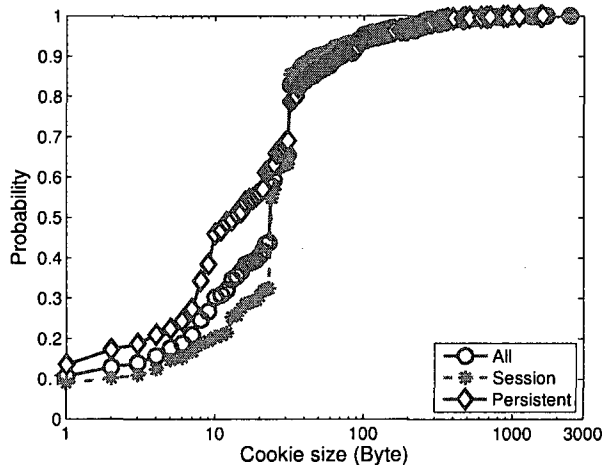


Figure 2.3: CDFs of cookie size in bytes for all cookies, session cookies, and persistent cookies, respectively.

states. Wide-spread indiscriminate usage of cookies not only impedes many optimizations such as content delivery optimizations studied in [9], but also increases the risk of cookie stealing.

We are especially interested in the lifetime of persistent cookies, which can be calculated from the cookie attribute `Max-Age` specified in the `Set-Cookie` HTTP response header. We compute the lifetime for all persistent cookies and draw its CDF in Figure 2.5 (The X-axis is log-scaled). There are a few hives in the curve, among which the lifetimes corresponding to one year (365 days) and 30 years are most evident. Clearly, over 60% of persistent cookies are set to expire after one year or longer. We also found that seven websites set their cookies to expire in year 9999, nearly eight thousand years from now! The mean and median lifetimes of persistent cookies are 38 years and one year, respectively.

In summary, our major measurement findings show that cookie has been widely used by websites. Although the simple cookie collection tool we used cannot retrieve the webpages that require login and cannot acquire the cookies set by JavaScript, we still found that about

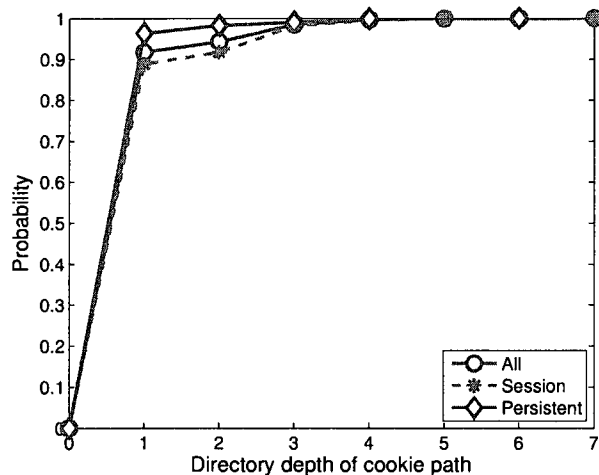


Figure 2.4: CDFs of cookie PATH depth for all cookies, session cookies, and persistent cookies, respectively.

43% of the total selected websites use either session cookies, persistent cookies, or both. Moreover, around 21% of the total selected websites use persistent cookies and the majority of persistent cookies have their lifetimes longer than one year. Therefore, the pervasive usage of persistent cookies and their very long lifetimes clearly highlight the demand for removing those useless persistent cookies to reduce potential privacy risks.

We did a similar measurement study in December 2006 [126]. The procedure and tool for selecting and crawling websites used in that study are the same as our current study. Although we cannot directly compare the results of two studies due to the differences of websites and crawling paths, we found that the findings mentioned above hold in the 2006 study. For example, in the 2006 study, about 37% of the total (5,211) selected websites use cookies and about 18% of the total selected websites use persistent cookies. Similarly, more than 60% of persistent cookies have lifetimes longer than one year in the 2006 study.

Recently Tappenden and Miller [103] did an extensive study on cookie deployment and

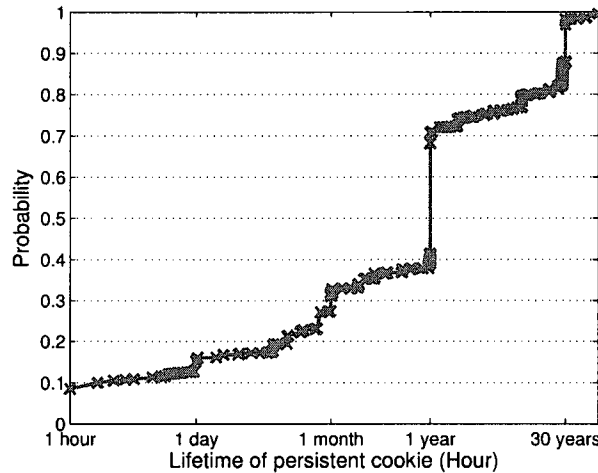


Figure 2.5: CDF of persistent cookie lifetime.

usage. Some of their study results have the similar implications as ours. For example, their study shows that 67.4% (66,031) of the 98,006 websites use cookies and 83.8% (55,130) of those 66,031 sites use first-party cookies. In addition, they observed that approximately 50.4% of all persistent cookies surveyed are set to live for over one year. The discrepancy in numbers between their study and ours can be attributed to the differences in the sample set and study methods. For example, they chose websites from Alexa while we chose websites from Google directory; they instrumented Firefox browser while we instrumented `wget`.

2.4 CookiePicker Design

The design goal of CookiePicker is to effectively identify the useful cookies of a website, and then disable the return of those useless cookies back to the website in the subsequent requests and finally remove them. A webpage is automatically retrieved twice by enabling and disabling some cookies. If there are obvious differences between the two retrieved results, we classify the cookies as useful; otherwise, we classify them as useless. CookiePicker

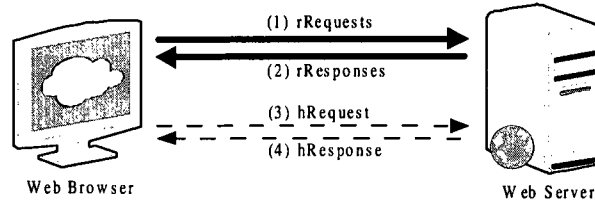


Figure 2.6: HTTP requests/responses in a single webpage view.

enhances the cookie management for a website by two processes: forward cookie usefulness marking and backward error recovery. We define these two processes and detail the design of CookiePicker in the following.

Definition 1. *FORward Cookie Usefulness Marking (FORCUM) is a training process, in which CookiePicker determines cookie usefulness and marks certain cookies as useful for a website.*

Definition 2. *Backward error recovery is a tuning process, in which wrong decisions made by CookiePicker in the FORCUM process may be adjusted automatically or manually for a website.*

2.4.1 Regular and Hidden Requests

A typical webpage consists of a container page that is an HTML text file, and a set of associated objects such as stylesheets, embedded images, scripts, and so on. When a user browses a webpage, the HTTP request for the container page is first sent to the Web server. Then, after receiving the corresponding HTTP response for the container page, the Web browser analyzes the container page and issues a series of HTTP requests to the Web server for downloading the objects associated with the container page. The HTTP requests and responses associated with a single webpage view are depicted by the solid lines (1) and (2)

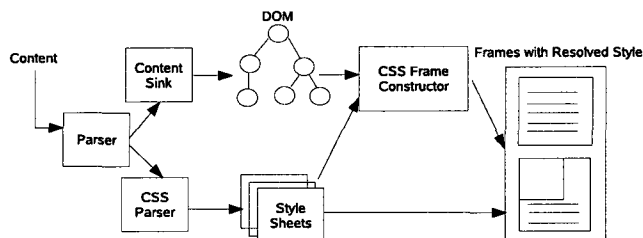


Figure 2.7: Data flow inside Gecko.

in Figure 2.6, respectively.

Webpage contents coming with the HTTP responses are passed into the Web browser layout engine for processing. Figure 2.7 depicts the data flow inside of Gecko [143], one of the most popular Web browser layout engines used in all Mozilla-branded software and its derivatives. When a webpage comes into Gecko, its container page is first parsed and built into a tree. The structure of the tree follows the W3C Document Object Model (DOM) [191], thus the tree is known as DOM tree. Next, the data from the DOM tree are put into abstract boxes called frames by combining the information from stylesheets. Finally, these frames and the associated Web objects are rendered and displayed on a user's screen.

In order to identify the cookie usefulness for a webpage, CookiePicker compares two versions of the same webpage: the first version is retrieved with cookies enabled and the second version is retrieved with cookies disabled. The first version is readily available to CookiePicker in the user's regular Web browsing window. CookiePicker only needs to retrieve the second version of the container page. Similar to Doppelganger [97], CookiePicker utilizes the ever increasing client side spare bandwidth and computing power to run the second version. However, unlike Doppelganger, CookiePicker neither maintains a fork window nor mirrors the whole user session. CookiePicker only retrieves the second version of

the container page by sending a single hidden HTTP request. As shown in Figure 2.6, line (3) is the extra hidden HTTP request sent by CookiePicker for the second version of the container page, and line (4) represents the corresponding HTTP response. In the remainder of the chapter, we simply refer the requests and responses, represented by the solid lines (1) and (2) of Figure 2.6, as regular requests and responses; and refer the extra request and response, represented by the dashed lines (3) and (4) of Figure 2.6, as the hidden request and response.

2.4.2 Forward Cookie Usefulness Marking

As shown in Figure 2.8, the FORCUM process consists of five steps: regular request recording, hidden request sending, DOM tree extraction, cookie usefulness identification, and cookie record marking.

When visiting a webpage, a user issues regular requests and then receives regular responses. At the first step, CookiePicker identifies the regular request for the container page and saves a copy of its URI and header information. CookiePicker needs to filter out the temporary redirection or replacement pages and locate the real initial container document page.

At the second step, CookiePicker takes advantage of user's think time [69] to retrieve the second copy of the container page, without causing any delay to the user's regular browsing. Specifically, right after all the regular responses are received and the webpage is rendered on the screen for display, CookiePicker issues the single hidden request for the second copy of the container page. In the hidden request, CookiePicker uses the same URI as the saved in the first step. It only modifies the "Cookie" field of the request header by removing a

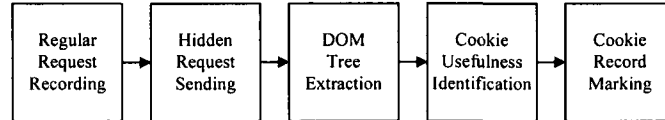


Figure 2.8: The five-step FORCUM process.

group of cookies, whose usefulness will be tested. The hidden request can be transmitted in an asynchronous mode so that it will not block any regular browsing functions. Then, upon the arrival of the hidden response, an event handler will be triggered to process it. Note that the hidden request is only used to retrieve the container page, and the received hidden response will not trigger any further requests for downloading the associated objects. Retrieving the container page only induces very low overhead to CookiePicker.

At the third step, CookiePicker extracts the two DOM trees from the two versions of the container page: one for the regular response and the other for the hidden response. We call these two DOM trees the regular DOM tree and the hidden DOM tree, respectively. The regular DOM tree has already been parsed by Web browser layout engine and is ready for use by CookiePicker. The hidden DOM tree, however, needs to be built by CookiePicker; and CookiePicker should build the hidden DOM tree using the same HTML parser of the Web browser. This is because in practice HTML pages are often malformed. Using the same HTML parser guarantees that the malformed HTML pages are treated as same as before, while the DOM tree is being constructed.

At the fourth step, CookiePicker identifies cookie usefulness by comparing the differences between the two versions of the container page, whose information is well represented in the two DOM trees. To make a right cookie usefulness decision, CookiePicker uses two complementary algorithms by considering both the internal structural difference and the

external visual content difference between the two versions. Only when obvious structural difference and visual content difference are detected, will CookiePicker decide that the corresponding cookies that are disabled in the hidden request are useful. The two algorithms will be detailed in Section 2.5.

At the fifth step, CookiePicker will mark the cookies that are classified as useful in the Web browser's cookie jar. An extra field "useful" is introduced to each cookie record. At the beginning of the FORCUM process, a *false* value is assigned to the "useful" field of each cookie. In addition, any newly-emerged cookies set by a website are also assigned *false* values to their "useful" fields. During the FORCUM process, the value of the field "useful" can only be changed in one direction, that is, from "false" to "true" if some cookies are classified as useful. Later on, when the values of the "useful" field for the existing cookies are relatively stable for the website, those cookies that still have "false" values in their "useful" fields will be treated as useless and will no longer be transmitted to the corresponding website. Then, the FORCUM process can be turned off for a while; and it will be turned on automatically if CookiePicker finds new cookies appeared in the HTTP responses or manually by a user if the user wants to continue the training process.

2.4.3 Backward Error Recovery

In general, CookiePicker could make two kinds of errors in the FORCUM process. The first kind of error is that useless cookies are misclassified as useful, thereby being continuously sent out to a website. The second kind of error is that useful cookies are never identified by CookiePicker during the training process, thereby being blocked from a website.

The first kind of error is solely due to the inaccuracy of CookiePicker in usefulness

identification. Such an error will not cause any immediate trouble to a user, but leave useless cookies increasing privacy risks. CookiePicker is required to make such errors as few as possible so that a user's privacy risk is lowered. CookiePicker meets this requirement via accurate decision algorithms.

The second kind of error is caused by either a wrong usefulness decision or the fact that some cookies are only useful to certain webpages but have not yet been visited during the FORCUM process. This kind of error will cause inconvenience to a user and must be fixed by marking the corresponding cookies as useful. CookiePicker attempts to achieve a very low rate on this kind of error, so that it does not cause any inconvenience to users. This requirement is achieved by two means. On one hand, for those visited pages, the decision algorithms of CookiePicker attempt to make sure that each useful persistent cookie can be identified and marked as useful. On the other hand, since CookiePicker is designed with very low running cost, a longer running period (or periodically running) of the FORCUM process is affordable, thus training accuracy can be further improved.

CookiePicker provides a simple recovery button for backward error recovery in the tuning process. In case a user notices some malfunctions or some strange behaviors on a webpage, the cookies disabled by CookiePicker in this particular webpage view can be re-marked as useful via a simple button click. Note that once the cookie set of a website becomes stable after the training and tuning processes, those disabled useless cookies will be removed from the Web browser's cookie jar. CookiePicker also provides an user interface, allowing a user to view those useless cookies and confirm the deletion action. We will introduce this interface in Section 2.6.

2.5 HTML Page Difference Detection

In this section, we present two complementary mechanisms for online detecting the HTML webpage differences between the enabled and disabled cookie usages. In the first mechanism, we propose a restricted version of Simple Tree Matching algorithm [117] to detect the HTML document structure difference. In the second mechanism, we propose a context-aware visual content extraction algorithm to detect the HTML page visual content difference. We call these two mechanisms as Restricted Simple Tree Matching (RSTM) and Context-aware Visual Content Extraction (CVCE), respectively. Intuitively, RSTM focuses on detecting the internal HTML document structure difference, while CVCE focuses on detecting the external visual content difference perceived by a user. In the following, we present these two mechanisms and explain how they are complementarily used.

2.5.1 Restricted Simple Tree Matching

As mentioned in Section 2.4, in a user’s Web browser, the content of an HTML webpage is naturally parsed into a DOM tree before it is rendered on the screen for display. Therefore, we resort to the classical measure of *tree edit distance* introduced by Tai [101] to quantify the difference between two HTML webpages. Since the DOM tree parsed from the HTML webpage is rooted (document node is the only root), labeled (each node has node name), and ordered (the left-to-right order of sibling nodes is significant), we only consider *rooted labeled ordered tree*. In the following, we will first review the tree edit distance problem; then we will explain why we choose *top-down distance* and detail the RSTM algorithm; and finally we will use Jaccard similarity coefficient to define the similarity metric of a

normalized DOM tree.

2.5.1.1 Tree Edit Distance

For two rooted labeled ordered trees T and T' , let $|T|$ and $|T'|$ denote the numbers of nodes in trees T and T' , and let $T[i]$ and $T'[j]$ denote the i th and j th preorder traversal nodes in trees T and T' , respectively. Tree edit distance is defined as the minimum cost sequence of edit operations to transform T into T' [101]. The three edit operations used in transformation include: inserting a node into a tree, deleting a node from a tree, and changing one node of a tree into another node. Disregarding the order of the edit operations being applied, the transformation from T to T' can be described by a mapping. The formal definition of a mapping [101] is as follows:

Definition 3. A mapping from T to T' is defined by a triple (M, T, T') , where M is any set of pairs of integers (i, j) satisfying:

$$(1) 1 \leq i \leq |T|, 1 \leq j \leq |T'|$$

(2) For any two pairs (i_1, j_1) and (i_2, j_2) in M ,

$$(a) i_1 = i_2 \text{ iff } j_1 = j_2;$$

$$(b) i_1 < i_2 \text{ iff } j_1 < j_2;$$

(c) $T[i_1]$ is an ancestor (descendant) of $T[i_2]$ iff

$$T'[j_1] \text{ is an ancestor (descendant) of } T'[j_2].$$

Intuitively, condition (2a) ensures that each node of both trees appears at most once in the mapping, and condition (2b) and (2c) ensure that the structural order is preserved in the mapping.

The algorithm presented by Tai [101] solves the tree edit distance problem in time $O(|T| \cdot |T'| \cdot |D|^2 \cdot |D'|^2)$, where $|D|$ and $|D'|$ are the maximum depths, respectively, of T and T' . Zhang *et al.* [130] further improved the result via a simple fast dynamic programming algorithm in time $O(|T| \cdot |T'| \cdot \min\{|D|, |L|\} \cdot \min\{|D'|, |L'|\})$, where $|L|$ and $|L'|$ are the numbers of leaves in T and T' , respectively.

Since the solution of the generic tree edit distance problem has high time complexity, researchers have investigated the constrained versions of the problem. By imposing conditions on the three edit operations mentioned above, a few different tree distance measures have been proposed and studied in the literature: *alignment distance* [50], *isolated subtree distance* [102], *top-down distance* [96, 117], and *bottom-up distance* [105]. The description and comparison of these algorithms can be found in [10] and [105].

2.5.1.2 Top-Down Distance

Because RSTM belongs to the top-down distance approach, we review the definition of top-down distance and explain why we choose this measure for our study.

Definition 4. A mapping (M, T, T') from T to T' , is top-down if it satisfies the condition that for all i, j such that $T[i]$ and $T'[j]$ are not the roots, respectively, of T and T' :

$$\text{if pair } (i, j) \in M \text{ then } (\text{parent}(i), \text{parent}(j)) \in M.$$

The top-down distance problem was introduced by Selkow [96]. In [117], Yang presented a $O(|T| \cdot |T'|)$ time-complexity top-down dynamic programming algorithm, which is named as the Simple Tree Matching (STM) algorithm. As we mentioned earlier, our goal is to

```

Algorithm: RSTM( $A, B, level$ )
1. if the roots of tree  $A$  and tree  $B$  contain different symbols then
2.   return(0);
3. endif
4.  $currentLevel = level + 1$ ;
5. if  $A$  and  $B$  are leaf or non-visible nodes or
6.    $currentLevel > maxLevel$  then
7.     return(0);
8. endif
9.  $m =$  the number of first-level subtrees of  $A$ ;
10.  $n =$  the number of first-level subtrees of  $B$ ;
11. Initialization,  $M[i, 0] = 0$  for  $i = 0, \dots, m$ ;
12.    $M[0, j] = 0$  for  $j = 0, \dots, n$ ;
13. for  $i = 1$  to  $m$  do
14.   for  $j = 1$  to  $n$  do
15.      $M[i, j] = \max(M[i, j - 1], M[i - 1, j],$ 
16.        $M[i - 1, j - 1] + W[i, j]);$ 
17.     where  $W[i, j] = \text{RSTM}(A_i, B_j, currentLevel)$ 
18.     where  $A_i$  and  $B_j$  are the  $i$ th and  $j$ th
19.     first-level subtrees of  $A$  and  $B$ , respectively
20.   endfor
21. endfor
22. return ( $M[m, n] + 1$ );

```

Figure 2.9: The restricted simple tree matching algorithm.

effectively detect noticeable HTML webpage difference between the enabled and disabled cookie usages. The measure of top-down distance captures the key structure difference between DOM trees in an accurate and efficient manner, and fits well to our requirement. In fact, top-down distance has been successfully used in a few Web-related projects. For example, Zhai and Liu [128] used it for extracting structured data from webpages; and Reis *et al.* [89] applied it for automatic Web news extraction. In contrast, bottom-up distance [105], although can be more efficient in time complexity ($O(|T| + |T'|)$), falls short of being an accurate metric [104] and may produce a far from optimal result [4] for HTML DOM tree comparison, in which most of the differences come from the leaf nodes.

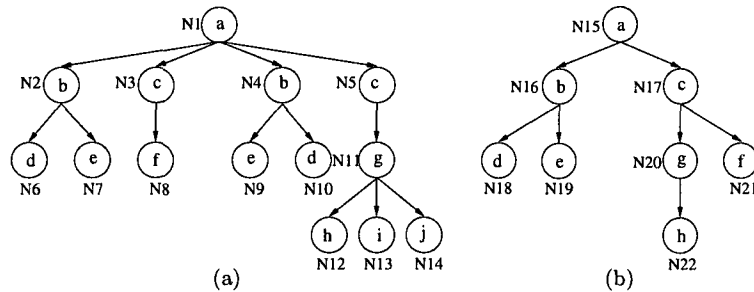


Figure 2.10: (a) Tree A, (b) Tree B.

2.5.1.3 Restricted Simple Tree Matching

Based on the original STM algorithm [117], Figure 2.9 illustrates RSTM, our restricted version of STM algorithm. Other than lines 4 to 8 and one new parameter *level*, our RSTM algorithm is similar to the original STM algorithm. Like the original STM algorithm, we first compare the roots of two trees A and B . If their roots contain different symbols, then A and B do not match at all. If their roots contain same symbols, we use dynamic programming to recursively compute the number of pairs in a maximum matching between trees A and B . Figure 2.10 gives two trees, in which each node is represented as a circle with a single letter inside. According to the preorder traversal, the fourteen nodes in tree A are named from $N1$ to $N14$, and the eight nodes in tree B are named from $N15$ to $N22$. The final result returned by STM algorithm or RSTM algorithm is the number of matching pairs for a maximum matching. For example, STM algorithm will return “7” for the two trees in Figure 2.10, and the seven matching pairs are $\{N1, N15\}$, $\{N2, N16\}$, $\{N6, N18\}$, $\{N7, N19\}$, $\{N5, N17\}$, $\{N11, N20\}$, and $\{N12, N22\}$.

There are two reasons why a new parameter *level* is introduced in RSTM. First, some webpages are very dynamic. From the same website, even if a webpage is retrieved twice

in a short time, there may exist some differences between the retrieved contents. For example, if we refresh Yahoo home page twice in a short time, we can often see some different advertisements. For CookiePicker, such dynamics on a webpage are just noises and should be differentiated from the webpage changes caused by the enabled and disabled cookie usages. The advertisement replacements on a webpage use different data items (e.g., images or texts) but they often stay at the same location of a webpage's DOM tree. Data items are mainly represented by lower level nodes of a DOM tree [129]. In contrast, the webpage changes caused by enabling/disabling cookies may introduce structural dissimilarities at the upper level of a DOM tree, especially when the theme of the page is changed. By using the new parameter *level*, the RSTM algorithm restricts the top-down comparison between the two trees to a certain maximum level. Therefore, equipped with the parameter *level*, RSTM not only captures the key structure dissimilarity between DOM trees, but also reduces leaf-level noises.

The second reason of introducing the new parameter *level* is that the $O(|T| \cdot |T'|)$ time complexity of STM is still too expensive to use online. Even with C++ implementation, STM will spend more than one second in difference detection for some large webpages. However, as shown in Section 2.6, the cost of the RSTM algorithm is low enough for online detection.

The newly-added conditions at line 5 of the RSTM algorithm restrict that the mapping counts only if the compared nodes are not leaves and have visual effects. More specifically, all the comment nodes are excluded in that they have no visual effect on the displayed webpage. Script nodes are also ignored because normally they do not contain any visual elements either. Text content nodes, although very important, are also excluded due to

the fact that they are leaf nodes (i.e., having no more structural information). Instead, text content will be analyzed in our Context-aware Visual Content Extraction (CVCE) mechanism.

2.5.1.4 Normalized Top-Down Distance Metric

Since the return result of RSTM (or STM) is the number of matching pairs for a maximum matching, based on the Jaccard similarity coefficient that is given in Formula 2.1, we define the normalized DOM tree similarity metric in Formula 2.2.

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \quad (2.1)$$

$$NTreeSim(A, B, l) = \frac{RSTM(A, B, l)}{N(A, l) + N(B, l) - RSTM(A, B, l)} \quad (2.2)$$

The Jaccard similarity coefficient $J(A, B)$ is defined as the ratio between the size of the *intersection* and the size of the *union* of two sets. In the definition of our normalized DOM tree similarity metric $NTreeSim(A, B, l)$, $RSTM(A, B, l)$ is the returned number of matched pairs by calling RSTM on trees A and B for upper l levels. $N(A, l)$ and $N(B, l)$ are the numbers of non-leaf visible nodes at upper l levels of trees A and B , respectively. Actually $N(A, l) = RSTM(A, A, l)$ and $N(B, l) = RSTM(B, B, l)$, but $N(A, l)$ and $N(B, l)$ can be computed in $O(n)$ time by simply preorder traversing the upper l levels of trees A and B , respectively.

2.5.2 Context-aware Visual Content Extraction

The visual contents on a webpage can be generally classified into two groups: text contents and image contents. Text contents are often displayed as headings, paragraphs, lists, table items, links, and so on. Image contents are often embedded in a webpage in the form

of icons, buttons, backgrounds, flashes, video clips, and so on. Our second mechanism mainly uses text contents, instead of image contents, to detect the visual content difference perceived by users. Two reasons motivate us to use text contents rather than image contents. First, text contents provide the most important information on webpages. This is because HTML mainly describes the structure of text-based information in a document, while image contents often serve as supplements to text contents [154]. In practice, users can block the loading of various images and browse webpages in text mode only. Second, the similarity between images cannot be trivially compared, while text contents can be extracted and compared easily as shown below.

On a webpage, each text content exists in a special context. Corresponding to the DOM tree, the text content is a leaf node and its context is the path from the root to this leaf node. For two webpages, by extracting and comparing their context-aware text contents that are essential to users, we can effectively detect the noticeable HTML webpage difference between the enabled and disabled cookie usages. Figure 2.11 depicts the recursive algorithm to extract the text content.

The `contentExtract` algorithm traverses the whole DOM tree in preorder in time $O(n)$. During the preorder traversal, each non-noise text node is associated with its context, resulting in a context-content string; and then the context-content string is added into set S . The final return result is set S , which includes all the context-content strings. Note that in lines 2 to 4, only those non-noise text nodes are qualified for the addition to set S . Similar to [38], scripts, styles, obvious advertisement text, date and time strings, and option text in dropdown lists (such as country list or language list) are regarded as noises. Text nodes that contain no alphanumeric characters are also treated as noises. All these

```

Algorithm: contentExtract( $T, context$ )
1. Initialization,  $S = \emptyset$ ;  $node = T.root$ ;
2. if  $node$  is a non-noise text node then
3.    $cText = context + SEPARATOR + node.value$ ;
4.    $S = S \cup \{cText\}$ ;
5. elseif  $node$  is an element node then
6.    $currentContext = context + ":" + node.name$ ;
7.    $n =$  the number of first-level subtrees of  $T$ ;
8.   for  $j = 1$  to  $n$  do
9.      $S = S \cup contentExtract(T_i, currentContext)$ ;
       where  $T_i$  is the  $i$ th first-level subtrees of  $T$ ;
10.  endfor
11. endif
12. return ( $S$ );

```

Figure 2.11: The text content extraction algorithm.

checkings guarantee that we can extract a relatively concise context-content string set from the DOM tree.

Assume S_1 and S_2 are two context-content string sets extracted from two DOM trees A and B , respectively. To compare the difference between S_1 and S_2 , again based on the Jaccard similarity coefficient, we define the normalized context-content string set similarity metric in Formula 2.3:

$$NTextSim(S_1, S_2) = \frac{|S_1 \cap S_2| + s}{|S_1 \cup S_2|} \quad (2.3)$$

Formula 2.3 is a variation [52] of the original Jaccard similarity coefficient. The extra added s on the numerator stands for the number of those context-content strings that are not exactly same, while having the same context prefix, in S_1 and S_2 . Intuitively, between two sets S_1 and S_2 , Formula 2.3 disregards the difference caused by text content replacement occurred in the same context, it only considers the difference caused by text content appeared in each set's unique context. This minor modification is especially helpful

in reducing the noises caused by advertisement text content and other dynamically changing text contents.

2.5.3 Making Decision

As discussed above, to accurately identify useful cookies, CookiePicker has to differentiate the HTML webpage differences caused by webpage dynamics from those caused by disabling cookies. Assume that tree A is parsed from a webpage retrieved with cookies enabled and tree B is parsed from the same webpage with cookies disabled. CookiePicker examines these two trees by using both algorithms presented above. If the return results of $NTreeSim$ and $NTextSim$ are less than two tunable thresholds, $Thresh1$ and $Thresh2$, respectively, CookiePicker will make a decision that the difference is due to cookie usage. Figure 2.12 depicts the final decision algorithm.

Note that these two thresholds are internal to CookiePicker, so a regular user does not need to know them. In our experiments (Section 2.6), we set the values of both thresholds to 0.85, and we found that no backward error recover is needed. We would like to recommend this as a reference value for CookiePicker. However, it is possible to further tune these two thresholds. For example, one approach is to self-adaptively adjust the thresholds based on the number or frequency of a user's backward error recover actions. The bottom line is that backward error recovery should not cause too much inconvenience to a user. Another approach is to use site-specific thresholds so that the granularity of accuracy can be refined to the site-level. In addition, it is also possible to allow users to share fine-tuned site-specific thresholds.

```
Algorithm: decision( $A, B, l$ )  
1. if NTreeSim( $A, B, l$ )  $\leq$  Thresh1 and  
2.   NTextSim( $S_1, S_2$ )  $\leq$  Thresh2 then  
3.   return the difference is caused by cookies;  
4. else  
5.   return the difference is caused by noises;  
6. endif
```

Figure 2.12: CookiePicker decision algorithm.

2.6 System Evaluation

In this section, we first briefly describe the implementation of CookiePicker, and then we validate its efficacy through two sets of live experiments.

2.6.1 Implementation

We implemented CookiePicker as a Firefox extension. Being one of the most popular Web browsers, Firefox is very extensible and allows programmers to add new features or modify existing features. Our CookiePicker extension is implemented in about 200 lines of XML user interface definition code, 1,600 lines of JavaScript code, and 600 lines of C++ code. JavaScript code is used for HTTP request/response monitoring and processing, as well as cookies record management. The HTML page difference detection algorithms are implemented in C++, because JavaScript version runs very slow. C++ code is compiled into a shared library in the form of an XPCOM (Cross-Platform Component Object Mode) component, which is accessible to JavaScript code. CookiePicker is a pure Firefox extension and it does not make any change to the Firefox's source code.

We omit other details and only describe two key interfaces in CookiePicker's implementation: the interface to user and the XPCOM component interface. Figure 2.13 shows

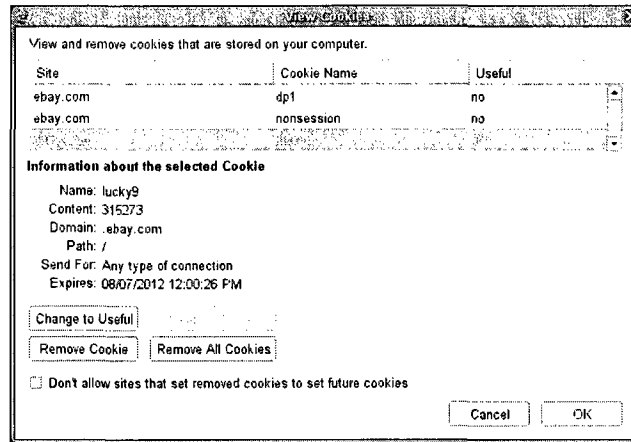


Figure 2.13: CookiePicker user interface.

CookiePicker’s main user interface. We port the code of a popular Firefox extension Cookie Culler [140] and merge them into CookiePicker. Cookie Culler allows a user to access cookie records and manually delete those cookies the user does not need to keep. By integrating this interface, CookiePicker provides a user with the capability to easily view the decisions made by CookiePicker and double check those useless cookies before they are finally removed from a browser’s cookie jar. As shown in Figure 2.13, if a user wants, the user can know that these three first-party persistent cookies from ebay.com have been automatically marked as useless and will be deleted by CookiePicker. The user can also make corrections to this result if necessary.

In the XPCOM component interface, two functions are defined as follows and they correspond to CookiePicker’s two HTML page difference detection algorithms, respectively:

```
interface ICookiePickerComponent : nsISupports {

float nTreeSim(in nsIDOMNode rNode, in nsIDOMNode hNode, in long l);

```

```

float nTextSim(in nsIDOMNode rNode, in nsIDOMNode hNode);
}

```

These two functions are implemented in C++, and can be accessed by JavaScript code via the `ICookiePickerComponent` component. For `nTreeSim`, its three input parameters match exactly with those in Figure 2.12. For `nTextSim`, its definition here is a little bit different from that in Figure 2.12, because the DOM trees are directly passed in and the corresponding context-content string sets are extracted internally.

2.6.2 Evaluation

We installed `CookiePicker` on a Firefox version 1.5.0.8 Web browser³ and designed two sets of experiments to validate the effectiveness of `CookiePicker` in identifying the useful first-party persistent cookies. The first set of experiments is to measure the overall effectiveness of `CookiePicker` and its running time in a generic environment; while the second set of experiments focuses on the websites whose persistent cookies are useful only, and examines the identification accuracy of `CookiePicker` upon useful persistent cookies. For all the experiments, the regular browsing window enables the use of persistent cookies, while the hidden request disables the use of persistent cookies by filtering them out from HTTP request header. The two thresholds used in `CookiePicker` decision algorithm are both set to 0.85, i.e., $Thresh1=Thresh2=0.85$. The parameter l for `NTreeSim` algorithm is set to 5, i.e., the top five level of DOM tree starting from the *body* HTML node will be compared by

³Porting `CookiePicker` to recent Firefox versions is quite feasible because `CookiePicker` uses the standard XPCOM mechanism of Firefox.

NTreeSim algorithm.

2.6.2.1 First Set of Experiments

From each of the 15 categories listed in Table 2.1, we randomly choose two websites that use persistent cookies. Thus, in total there are 30 websites in the first set of experiments. As listed in the first column of Table 2.2, these 30 websites are represented as S1 to S30 for privacy concerns.

Inside each website, we first visit over 25 webpages to stabilize its persistent cookies and the “useful” values of the persistence cookies, i.e, no more persistent cookies of the website are marked as “useful” by CookiePicker afterwards. Then, we count the number of persistent cookies set by the website and the number of persistent cookies marked as useful by CookiePicker. These two numbers are shown in the second and third columns of Table 2.2, respectively. Among the total 30 websites, the persistent cookies from five websites (S1,S6,S10,S16,S27) are marked as “useful” by CookiePicker, and the persistent cookies from the rest of 30 websites are identified as “useless”. In other words, CookiePicker indicates that we can disable the persistent cookies in about 83.3% (25 out of 30) of testing websites. To further validate the testing result above, we check the uselessness of the persistent cookies for those 25 websites through careful manual verification. We find that blocking the persistent cookies of those 25 websites does not cause any problem to a user. Therefore, none of the classified “useless” persistent cookies is useful, and no backward error recovery is needed.

For those five websites that have some persistent cookies marked as “useful”, we verify the real usefulness of these cookies by blocking the use of them and then comparing the

Website	Persistent	Marked Useful	Real Useful	Detection Time(ms)	CookiePicker Duration (ms)
S1	2	2	0	8.3	1,821.6
S2	4	0	0	9.3	5,020.2
S3	5	0	0	14.8	1,427.5
S4	4	0	0	36.1	9,066.2
S5	4	0	0	5.4	698.9
S6	2	2	2	5.7	1,437.5
S7	1	0	0	17.0	3,373.2
S8	3	0	0	7.4	2,624.4
S9	1	0	0	13.2	1,415.4
S10	1	1	0	5.7	1,141.2
S11	2	0	0	2.7	941.3
S12	4	0	0	21.7	2,309.9
S13	1	0	0	8.0	614.9
S14	9	0	0	11.9	1,122.4
S15	2	0	0	8.5	948.0
S16	25	1	1	5.8	455.9
S17	4	0	0	7.5	11,426.3
S18	1	0	0	23.1	4,056.9
S19	3	0	0	18.0	3,860.5
S20	6	0	0	8.9	3,841.6
S21	3	0	0	14.4	936.1
S22	1	0	0	13.1	993.3
S23	4	0	0	28.8	2,430.1
S24	1	0	0	23.6	2,381.1
S25	3	0	0	30.7	550.1
S26	1	0	0	5.03	611.6
S27	1	1	0	8.7	597.5
S28	1	0	0	10.7	10,104.1
S29	2	0	0	7.7	1,387.1
S30	2	0	0	57.6	2,905.6
Total	103	7	3	-	-
Average	-	-	-	14.6	2,683.3

Table 2.2: Online testing results for thirty websites (S1 to S30).

disabled version with a regular browsing window over 25 webpages in each website. The result is shown in the fourth column of Table 2.2. We observe that three cookies from two websites (S6,S16) are indeed useful. However, for the other three websites (S1,S10,S27), their persistent cookies are useless but are wrongly marked as “useful” by CookiePicker. This is mainly due to the conservative threshold setting. Currently the values of both thresholds are set to 0.85, i.e., $Thresh1=Thresh2=0.85$. The rationale behind the conservative threshold setting is that we prefer to have all useful persistent cookies be correctly

identified, even at the cost of some useless cookies being misclassified as “useful”. Thus, the number of backward error recovery is minimized.

In Table 2.2, the fifth and sixth columns show the average running time of the detection algorithms and the entire duration of CookiePicker, respectively. It is clear that the running time of the page difference detection is very short with an average of 14.6 ms over the 30 websites. This detection time is roughly the extra CPU computation time introduced by running the CookiePicker browser extension. The extra memory resource consumption is also negligible because CookiePicker does not have a large memory requirement. The average identification duration is 2,683.3 ms, which is reasonable short considering the fact that the average think time of a user is about 10 seconds [69]. Note that websites S4, S17, and S28 have abnormally high identification duration at about 10 seconds, which is mainly caused by the slow responses from these websites.

2.6.2.2 Second Set of Experiments

Since only two websites in the first set of experiments have useful persistent cookies, we attempt to further examine if CookiePicker can correctly identify each useful persistent cookie in the second set of experiments. Because the list of websites whose persistent cookies are really useful to users does not exist, we have to locate such websites manually. Again, we randomly choose 200 websites that use persistent cookies from the 15 categories listed in Table 2.1. Note that the 30 websites chosen in the first set of experiments are not included in these 200 websites. We manually scrutinize these 200 websites, and finally find six websites whose persistent cookies are really useful to users, i.e., without cookies, users would encounter some problems. Because the manual scrutiny is tedious, we cannot afford

more effort to locate more such websites. The six websites are listed in the first column of Table 2.3 and represented as P1 to P6 for privacy concerns.

In Table 2.3, the second column shows the number of the cookies marked as “useful” by CookiePicker and the third column shows the number of the real useful cookies via manual verification. We observe that for the six websites, all of their useful persistent cookies are marked as “useful” by CookiePicker. This result indicates that CookiePicker seldom misses the identification of a real useful cookie. On the other hand, for websites P5 and P6, some useless persistent cookies are also marked as “useful” because they are sent out in the same regular request with the real useful cookies. The fourth and fifth columns show the similarity score computed by $NTreeSim(A, B, 5)$ and $NTextSim(S_1, S_2)$, respectively, on the webpages that persistent cookies are useful. These similarity scores are far below 0.85, which is the current value used for the two thresholds *Thresh1* and *Thresh2* in Figure 2.12. The usage of these useful persistent cookies on each website is given at the sixth column. Websites P1, P4, and P6 use persistent cookies for user’s preference setting. Websites P3 and P5 use persistent cookies to properly create and sign up a new user. Website P2 uses persistent cookie in a very unique way. Each user’s persistent cookie corresponds to a specific sub-directory on the Web server, and the sub-directory stores the user’s recent query results. Thus, if the user visits the website again with the persistent cookie, recent query results can be reused to improve query performance.

In summary, the above two sets of experiments show that by conservatively setting *Thresh1* and *Thresh2* to 0.85, CookiePicker can safely disable and remove persistent cookies from about 83.3% of websites (25 out of the 30 websites that we intensively tested). Meanwhile, all the useful persistent cookies are correctly identified by CookiePicker and no

Website	Marked Useful	Real Useful	NTreeSim ($A, B, 5$)	NTextSim (S_1, S_2)	Usage
P1	1	1	0.311	0.609	Preference
P2	1	1	0.459	0.765	Performance
P3	1	1	0.667	0.623	Sign Up
P4	1	1	0.250	0.158	Preference
P5	9	1	0.226	0.253	Sign Up
P6	5	2	0.593	0.719	Preference
Average	-	-	0.418	0.521	-

Table 2.3: Online testing results for 6 websites (P1 to P6) that have useful persistent cookies.

backward error recovery is needed for all the 8 websites (S6,S16,P1,P2,P3,P4,P5,P6) that have useful persistent cookies. Misclassification happens only in 10% websites (3 out of 30), on which useless persistent cookies are wrongly identified as useful.

2.7 Discussions

In CookiePicker, useful cookies are defined as those that can cause perceivable changes on a webpage, and as we discussed before this definition is probably the most reasonable measure right now at the browser side. We assume that this measure is known to anyone who wants to evade CookiePicker. In this section, we first identify possible evasion techniques, and then we analyze potential evasion sources and explain why those evasion techniques are not a serious concern to CookiePicker. Finally, we discuss some concerns about using CookiePicker.

2.7.1 Possible Evasion Techniques

Since CookiePicker makes its decision based on HTML page difference detection, we identify the following four possible techniques that could be used to evade CookiePicker:

- *random advertising*: A website can serve more random advertisements to different retrieves of a webpage in order to reduce the accuracy of CookiePicker’s difference detection algorithms.
- *dynamic content rewriting*: A website can identify CookiePicker’s hidden request and use JavaScript to dynamically rewrite webpage contents at the browser side.
- *dramatic structure changing*: A website can identify CookiePicker’s hidden request and intentionally generate structurally different response pages (even if with very similar visual content).
- *cookie root path abusing*: A website can set attribute values of all its cookies as root “/” to let all its cookies, no matter useful or useless, return back for every Web request.

For the *random advertising* evasion technique, indeed as shown in previous experimental results, CookiePicker’s two difference detection algorithms can filter out the effects of advertisements and webpage dynamics very well. Moreover, other advertisement removing techniques such as those used in [178] can be integrated into CookiePicker to further improve the detection accuracy. The other three kinds of evasion techniques can be employed by a website operator. However, as we discuss below, they will not pose a serious threat to the usage of CookiePicker.

2.7.2 Evasion Sources

The evasion of CookiePicker will most likely come from two sources: website operators who want to track user activities, and attackers who want to steal cookies. As stated in Section 2.2, we assume that the hosting website is legitimate, since it is pointless to provide

cookie security and privacy services for a malicious website. For legitimate websites, if some operators strongly insist to use first-party persistent cookies for tracking long-term user behaviors, they can evade CookiePicker by detecting the hidden HTTP request and manipulating the hidden HTTP response using the evasion techniques mentioned above. However, we argue that many website operators will not pay the effort and time to do so, either because of the lack of interest to track long-term user behaviors in the first place, or because of inaccuracy in cookie-based user behavior tracking, which has long been recognized [194]. These website operators are either not aware of the possible privacy and security risks of stolen first-party persistent cookies, or simply not willing to pay the cost to renovate their systems. Our CookiePicker is a pure client-side solution that especially aims to protect the users of these websites.

For third-party attackers, unless they compromise a legitimate website, it is very difficult for them to use any of the above evasion techniques to manipulate the webpages sending back to a user's browser and circumvent CookiePicker. Therefore, CookiePicker can effectively identify and remove useless cookies stored by most legitimate websites on a user's hard disk, and prevent them from being stolen by malicious attacks such as cross-site scripting attacks.

2.7.3 Concerns about Using CookiePicker

One concern is that CookiePicker might reject third-party cookies in the case of HTTP redirection (see Section 5.1.2 of [103]). However, this concern is invalid because CookiePicker does not block (or even touch) third-party cookies by itself - decision regarding third-party cookies is made completely by the Web browser based on users' preference setting. Similarly,

whether a cookie is a first-party cookie of a website is also decided by the Web browser, and CookiePicker makes its decision based on the browser's decision. Therefore, it does not matter whether a website is using CDN (Content Delivery Network) or not.

Another concern is that CookiePicker may not work well on webpages that use Ajax (Asynchronous JavaScript and XML). This concern is reasonable because Ajax requests may use cookies and may also change webpage contents. Currently, CookiePicker only sends hidden requests to retrieve container pages, as described in Section 2.4. But it is possible to extend CookiePicker to evaluate the impact of cookies on Ajax requests, for example, by asking CookiePicker to send out two Ajax requests (one with cookies and the other without cookies) at the same time and then compare the response messages.

2.8 Related Work

RFC 2109 [61] specifies the way of using cookies to create a stateful session with HTTP requests and responses. It is also the first document that raises the general public's awareness of cookie privacy problems. RFC 2965 [62] follows RFC 2109 by introducing two new headers, `Cookie2` request header and `Set-Cookie2` response header. However, these two new headers are not supported by the popular Web browsers such as Internet Explorer and Firefox. RFC 2964 [76] focuses on the privacy and security of using HTTP cookies, and identifies the specific usages of cookies that are either not recommended by the IETF or believed to be harmful. Fu's study [34] suggests that setting authenticators in cookies should be very careful and especially persistent cookies should not be used to store authenticators.

Cookies not only can be retrieved and stored by the headers of HTTP requests and

responses, but also can be read and written by client-side JavaScript. The *same origin policy* [180] introduced in Netscape Navigator 2.0 prevents cookies and JavaScript in different domains from interfering with each other. The successful fulfillment of the *same origin policy* on cookies and JavaScript further invokes the enforcement of this policy on browser cache and visited links [44]. Recently, in order to mitigate cross-site scripting attacks, Internet Explore also allows a cookie to be marked as “HttpOnly” in the `Set-Cookie` response header, indicating that a cookie is “non-scriptable” and should not be revealed to client applications [170].

Modern Web browsers have provided users with refined cookie privacy options. A user can define detailed cookie policies for websites either before or during visiting these sites. Commercial cookie management software such as Cookie Crusher [139] and CookiePal [141] mainly rely on pop-ups to notify incoming cookies. However, the studies in [39] show that such cookie privacy options and cookie management policies fail to be used in practice, due mainly to the following two reasons: (1) these options are very confusing and cumbersome, and (2) most users have no good understanding of the advantages and disadvantages of using cookies. A few Firefox extensions such as Cookie Culler [140] and Permit Cookies [175], although convenient to use, are just very simple add-ons for user to easily access privacy preference settings or view cookies. Acumen system [37] can inform a user how many other users accept certain cookies. However, the system does not protect the privacy of the user itself. Moreover, many users’ decisions could be wrong, resulting in negative reference. Another interesting system is Privoxy [178] Web proxy. It provides advanced filtering capabilities for protecting privacy, modifying webpage data, managing cookies, controlling access, and removing advertisement, banners, pop-ups and other obnoxious Internet junk.

However, Privoxy is more useful for those sophisticated users who have the ability to fine-tune their installation.

Recently, the most noticeable research work in cookie management is Doppelganger [97]. Doppelganger is a system for creating and enforcing fine-grained privacy-preserving cookie policies. Doppelganger leverages client-side parallelism and uses a twin window to mirror a user's Web session. If any difference is detected, Doppelganger will ask the user to compare the main window and the fork window, and then, make a cookie policy decision. Although taking a big step towards automatic cookie management, Doppelganger still has a few obvious drawbacks. First, Doppelganger still heavily relies on the user's comparison between the main window and the fork window to make a decision. Second, the cost of its parallel mirroring mechanism is very high. This is because not only every user action needs to be mirrored, but every HTTP request also needs to be dublicately sent back to the Web server. Third, due to the high cost, a user may not be patient enough to have a long training period, thus the policy decision accuracy cannot be guaranteed. Last but not the least, Doppelganger only achieves website level cookie policy making. In contrast, our CookiePicker works fully automatically without user involvement or even notice. It has very low overhead, and hence, can be trained for a long period on a user's Web browser to achieve high accuracy. CookiePicker achieves cookie-group level policy making, implying that usefulness is identified for a group of cookies used in a webpage view.

2.9 Summary

In this work, we conducted a large-scale cookie measurement, which highlights the demand for effective cookie management. Then, we presented a system, called CookiePicker, to automatically managing cookie usage setting on behalf of a user. Only one additional HTTP request for the container page of a website, the hidden request, is generated for CookiePicker to identify the usefulness of a cookie set. CookiePicker uses two complementary algorithms to accurately detect the HTML page differences caused by enabling and disabling cookies. CookiePicker classifies those cookies that cause perceivable changes on a webpage as useful, and disable the rest as useless. We implemented CookiePicker as an extension to Firefox and evaluated its efficacy through live experiments over various websites. By automatically managing the usage of cookies, CookiePicker helps a user to strike an appropriate balance between maximizing usability and minimizing security risks.

Chapter 3

Transparent Phishing Protection

In this chapter, we present our work on transparent phishing protection. Modern Web browsers are often equipped with automatic phishing detection and warning mechanisms. However, usability studies have demonstrated that many users still ignore the strong phishing warnings given by browsers and become victims. We present an approach that uses bogus username/password credentials to protect against phishing attacks. Seamlessly integrated with the phishing detection and warning mechanisms in modern Web browsers, our approach properly leverages existing anti-phishing efforts and is capable of providing a transparent protection to those most vulnerable users who still ignore the strong phishing warnings.

3.1 Motivation

A phishing attack is typically carried out using an email or an instant message, in an attempt to lure recipients to a fake website to disclose personal credentials. Phishing attacks

have seriously afflicted Internet users and financial institutions with identity thefts and brand reputation damage. According to recent Anti-Phishing Working Group (APWG) reports [137] and Gartner surveys [149], the number of phishing sites, the number of phishing victims, and the amount of financial losses stemming from phishing attacks have all increased over the past few years.

To defend against phishing attacks, a number of countermeasures have been proposed and developed. Server-side defenses employ SSL certificates, user-selected site images, and other security indicators to help users verify the legitimacy of websites. Client-side defenses equip Web browsers with automatic phishing detection features or add-ons to warn users away from suspected phishing sites. However, recent usability studies have demonstrated that neither server-side security indicators nor client-side toolbars and warnings are successful in preventing vulnerable users from being deceived [21, 20, 95, 111, 114, 23]. This is mainly because (1) phishers can convincingly imitate the appearance of legitimate websites, (2) users tend to ignore security indicators or warnings, and (3) users do not necessarily interpret security cues appropriately. Educational defenses teach users to understand and avoid phishing attacks [45, 64, 100]. However they cannot completely foil phishing attacks. Takedown defenses exploit spams and suspicious URLs to discover and shut down newly emerged phishing sites. However, the efficacy of this approach is limited, due to the ease of setting up and the short online time of phishing sites, as well as the application of takedown evasion methods by phishers [46, 77, 137, 165].

These different approaches are all preventive by nature. They endeavor to prevent users from being tricked into revealing their credentials to phishing sites. Nevertheless, these prevention-based approaches alone are insufficient to shield vulnerable users from “biting

the bait” and defeat phishers, as human users are the weakest link in the security chain. The ever-increasing prevalence and severity of phishing attacks clearly indicate that anti-phishing is still a daunting challenge.

In response to this challenge, we have made two observations with respect to the acquisition of credentials by phishers and the automatic detection of phishing attacks on Web browsers. First, currently the majority of those who have “bitten the bait” and fallen victim to phishing attacks are real victims; thus, it is trivial for a phisher to verify the acquired credentials and trade them for money. However, if we can supply phishing sites with a large number of bogus credentials, we might be able to hide victims’ real credentials among bogus credentials and make it harder for phishers to succeed.

Second, although remarkable advances in client-side automatic phishing detection have empowered Web browsers to identify the majority of phishing sites [17, 35, 68, 132, 148, 169], the possible false positives (legitimate websites misclassified as phishing sites) make it hard for Web browsers to directly block users’ connections to suspected phishing sites. Thus, issuing warnings and expecting users to leave a suspected phishing site have become the most common actions employed by modern Web browsers. However, instead of just wishing vulnerable users could make correct decisions, if we can effectively transform the power of automatic phishing detection into the power of automatic fraud protection, we will take a big step forward towards winning the battle against phishing.

In this work, we propose a new approach to protect against phishing attacks with “bogus bites” on the basis of the two observations we have mentioned. The key feature of this approach is to transparently feed a relatively large number of bogus credentials into a suspected phishing site, rather than attempt to prevent vulnerable users from “biting the

bait.” These “bogus bites” conceal victims’ real credentials among bogus credentials, and enable legitimate websites to identify stolen credentials in a timely manner. Based on the concept of “bogus bites,” we design and develop *BogusBiter*, a unique client-side anti-phishing tool that is complementary to existing prevention-based mechanisms. Seamlessly integrated with the phishing detection and warning mechanisms in modern Web browsers, BogusBiter is transparent to users.

At a user’s Web browser, BogusBiter is turned on once a login webpage is classified as a phishing page by a Web browser’s built-in phishing detection component or a third-party detection toolbar. For a victim who is beguiled into divulging a real credential, BogusBiter hides the real credential among a set of automatically generated bogus credentials, and then submits these credentials one by one to the phishing site. For a security-conscious user who does not reveal a real credential, BogusBiter also generates a set of bogus credentials, and then submits them to the phishing site in the same way as it does for a victim.

At the phishing site, a phisher will thus receive a much larger number of credentials than before, but the overwhelming majority are bogus credentials fed by BogusBiter. Elaborating bogus credential generation and submission mechanisms, BogusBiter makes it difficult for a phisher to distinguish who are real victims and which are real credentials. The only effective way for a phisher to sift out bogus credentials is to visit the legitimate website and verify whatever credentials have been collected from the phishing site.

At the legitimate website, if the phisher assumes the burden of verifying all the collected credentials to single out the real credentials, the unique design of bogus credential generation will enable the legitimate site to identify victims’ stolen credentials in a timely manner and make it harder for a phisher to succeed. In other words, the bogus credential filtering

process becomes the trigger for detecting stolen credentials at the legitimate website, and hence, ironically, the phisher's attempt to bypass BogusBiter helps us to achieve automatic fraud protection.

While leveraging the power of widely used client-side automatic phishing detection techniques, BogusBiter is not bound to any specific phishing detection scheme. Thus, BogusBiter can utilize the latest advances in phishing detection techniques such as blacklists and heuristics to protect against a wide range of phishing attacks. Moreover, BogusBiter is incrementally deployable over the Internet, and the fraud protection enabled at a legitimate website is independent of the deployment scale of BogusBiter. We implemented BogusBiter as a Firefox Web browser extension and evaluated its efficacy through real experiments over both phishing and legitimate websites. Our experimental results indicate that it is promising to use BogusBiter to transparently protect against phishing attacks.

The remainder of this chapter is structured as follows. Section 3.2 introduces the background of phishing attacks and the automatic phishing detection and warning mechanisms in modern Web browsers. Section 3.3 details the design of BogusBiter. Section 3.4 describes the implementation of BogusBiter. Section 3.5 evaluates the capability and performance of BogusBiter. Section 3.6 discusses the deployment of BogusBiter and potential evasions against BogusBiter. Section 3.7 reviews the related client-side anti-phishing research work, and finally, Section 3.8 summarizes this work.

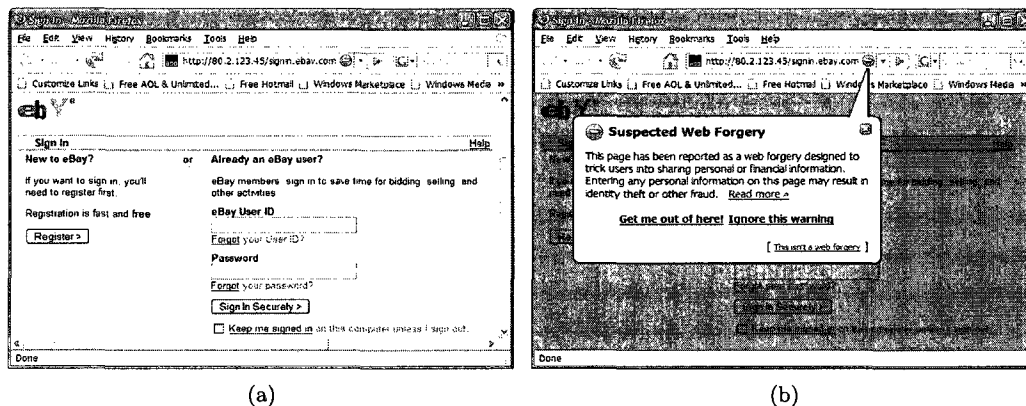


Figure 3.1: (a) A phishing site designed to attack eBay users, (b) Firefox 2 phishing warning mechanism.

3.2 Background

Figure 3.1(a) illustrates a phishing site designed to attack eBay users. In a typical scenario, a user receives a spoofed email that appears to be sent from the real eBay, luring the user to log into the phishing site. Once the user believes this site is the genuine eBay website and logs in, the user's username/password credential is stolen. Passwords have increasingly been targeted by harvesting attacks, as they protect online accounts with valuable assets [31]. While some phishing attacks may steal other types of credentials such as credit card numbers and social security numbers, the most common type of phishing attack attempts to steal account numbers and passwords used for online banking [48]. Therefore, protecting a user's username/password credential is the primary focus of many client-side anti-phishing research work such as SpoofGuard [17], Dynamic Security Skins [19], AntiPhish [56], Pwd-Hash [91], Web Wallet [115], and Passpet [119]. Our work also focuses on protecting a user's username/password credential. In the remainder of this chapter, we use the terms credential and username/password pair interchangeably.

The potential threat of phishing or Web spoofing attacks was first uncovered by Felten et al. [25]. Today, phishing is not merely about website forgery and email spoofing, it has become a carefully planned and well structured multiphase effort to steal money. The life cycle of a phishing attack consists of six phases: *planning*, *setup*, *attack*, *collection*, *fraud & abuse*, and *post attack*, as defined by the Financial Services Technology Consortium [188]. Various techniques can be applied in these six phases to combat phishing. However, since phishing sites can be easily set up and money laundering is still a difficult problem to curtail, current research and industry efforts focus mainly on the *attack* phase, with the objective of preventing users from submitting their credentials to phishing sites.

In contrast, the BogusBiter's protection against phishing attacks ranges from the *attack* phase to the *collection* phase, and then to the *fraud & abuse* phase, covering the flow of credentials. Specifically, BogusBiter retaliates against phishers with a large number of bogus credentials in the *attack* phase, makes it hard for them to identify real credentials in the *collection* phase, and detects their fraudulent activities in the *fraud & abuse* phase.

While distinct from preventive anti-phishing mechanisms, BogusBiter complements them in a natural way. In particular, BogusBiter leverages the power of client-side automatic phishing detection mechanisms and takes advantage of the state-of-practice phishing warning mechanisms in popular Web browsers to transparently protect vulnerable users.

Among automatic phishing detection mechanisms, two commonly used techniques are blacklists and heuristics. Blacklist-based techniques generate close-to-zero false positives and can detect most phishing attacks [68, 131, 146, 151]. For example, Ludl et al. [68] demonstrated that blacklists provided by Google (used by Firefox 2) can recognize almost 90% of live phishing sites. However, because some phishing sites may not be added

into blacklists and the so-called *zero-day* attacks may occur, researchers have proposed various heuristic-based techniques to identify phishing sites in real time [17, 35, 68, 132]. These heuristic-based techniques have obtained very encouraging results. For example, CANTINA, a content-based detection tool proposed by Zhang et al. [132] can identify 90% of phishing pages with only 1% false positives. A URL-based classifier proposed by Garera et al. [35] is another tool that can catch 95.8% of phishing pages with only 1.2% false positives.

Currently, Firefox 2 primarily employs blacklist-based techniques while Internet Explorer (IE) 7 uses both kinds of techniques [148, 169]. Because BogusBiter's design is independent of any specific detection scheme, it can leverage advances in both blacklist-based techniques and heuristic-based techniques to combat the majority of phishing attacks.

Regarding phishing site warning mechanisms, the state of practice is to make it mandatory for a user to respond to the active warning of a suspected phishing site. Figure 3.1(b) illustrates the warning given by Firefox 2 [148] after correctly identifying the example website in Figure 3.1(a) as a phishing site. A user is unable to enter the username and password without first interacting with the warning page. If the user clicks the "Get me out of here!" link, the user is redirected to a default page and is protected. Otherwise, if the user clicks the "Ignore this warning" link, the warning page disappears and the user is exposed to the risk of credential theft. A similar warning mechanism is also used in IE 7 [169].

Both Firefox 2 and IE 7 might choose such a active warning mechanism because: (1) issuing warnings simply through browser-based security indicators such as the address bar, the status bar, and various toolbars is ineffective [21, 20, 95, 111, 114, 23], and (2) directly blocking users' connections to suspected phishing sites is unacceptable, due to inevitable

false positives. Although using an active warning page represents current best practice, a recent usability study conducted by Egelman et al. [23] demonstrates that overall about 21% of participants still ignore the IE 7 and Firefox 2 active phishing warnings and fall for phishing attacks. Therefore, a crucial usability gap still exists in today's anti-phishing ecosystem, and many users who are most vulnerable to phishing still cannot be protected.

Phishing attacks are very insidious, and so far there is no single silver bullet for completely defeating phishers. Comprehensive, multifaceted, and integrated approaches are clearly needed in the anti-phishing ecosystem. BogusBiter fits into such an anti-phishing ecosystem especially by aiming to fill the aforementioned usability gap. Properly leveraging existing anti-phishing efforts, BogusBiter is capable of providing a transparent protection to those most vulnerable users.

3.3 Design

In this section, we first give an overview on the design of BogusBiter, including the basic working mechanism, the main design assumption, and the two key design objectives. We then detail the offensive line and defensive line of BogusBiter.

3.3.1 Design Overview

BogusBiter is designed as either a new component or an extension to popular Web browsers such as Firefox 2 or IE 7. It integrates seamlessly with phishing detection and warning mechanisms of current Web browsers to protect vulnerable users against phishing attacks.

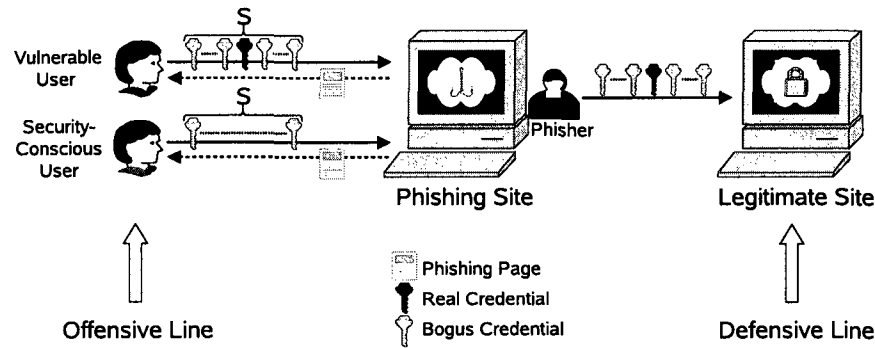


Figure 3.2: Anti-phishing with BogusBiter.

3.3.1.1 How It Works

In the scenario without BogusBiter, when a phishing site is visited by users, only real credentials are submitted by vulnerable users, and a phisher can easily verify the collected credentials and trade them for money.

The basic idea of BogusBiter is very simple, as illustrated in Figure 3.2. When a login page is classified as a phishing page by a browser's built-in detection component or a third-party detection toolbar, BogusBiter is triggered. At this point, BogusBiter will perform differently based on a user's response to the browser's phishing warning page. For a vulnerable user who clicks the "Ignore this warning" link and submits a real credential, BogusBiter will intercept the victim's real credential, hide it among a set of $S - 1$ generated bogus credentials, and then submit the S credentials one by one to the phishing site within a few milliseconds. For a security-conscious user who clicks the "Get me out of here!" link on the warning page, BogusBiter will generate a set of S bogus credentials, and then feed them one by one into the phishing site in the same way as it does for a vulnerable user. These actions are completely transparent to both vulnerable and security-conscious users.

The BogusBiter extensions installed on users' browsers make up the offensive line. Later on, when a phisher verifies the collected credentials at the legitimate site, the defensive line enabled by BogusBiter will help a legitimate site to detect victims' stolen credentials in a timely manner.

3.3.1.2 Design Assumption

We assume that a phisher does not have a complete list of valid usernames for a targeted legitimate website, and cannot directly query a targeted legitimate website for the validity of a specific username. Although this assumption may not be strictly correct for email service websites and community websites, it is generally true for financial institutions, which are the main targets of phishing attacks. Financial institutions seldom have valid username lists publicly accessible. Meanwhile, for a failed login attempt, websites often try to hide whether the failure is due to an incorrect username or due to an incorrect password by returning the same error message [12, 32], making it very hard to test the validity of a given username.

Indeed, preventing the leakage of username validity information is necessary for protecting user privacy, guarding users from invasive advertising and phishing, and defending against password guessing attacks. To enhance such a protection, the recent work by Bortz et al. [12] recommends that the response time of HTTP requests should be carefully controlled by some websites to remove timing vulnerabilities. Florêncio et al. [32] further suggest that increasing username strength could be more beneficial than merely increasing password strength.

3.3.1.3 Design Objectives

To be effective, BogusBiter has two key design objectives:

- *offensive objective*: BogusBiter should inject as many bogus credentials as possible into a phishing site, thus well hiding victims' real credentials among bogus credentials.
- *defensive objective*: Given that a phisher is aware of BogusBiter and is willing to assume the heavy burden of sifting out bogus credentials, BogusBiter should enable a legitimate website to exploit the filtering process initiated by the phisher to detect victims' stolen credentials in a timely manner.

3.3.2 Offensive Line

To achieve its *offensive objective*, BogusBiter should strive to meet the following three requirements.

- *Massiveness*: The number of bogus credentials fed into a phishing site should be large so that the overwhelming majority of credentials received by a phisher are bogus.
- *Indiscernibility*: Without the credential verification at the legitimate website, it is extremely difficult for a phisher to deterministically discern, either at credential submission time or afterwards, who are real victims and what are real credentials.
- *Usability*: The usage of BogusBiter at the client-side should not incur undue overhead or unwanted side effects, nor should it produce any security or privacy concerns.

3.3.2.1 Massiveness

We use the *real-to-all* ratio—the ratio between the number of real credentials being collected and the total number of credentials being collected—to estimate how many bogus credentials should be fed into a phishing site to hide victims’ real credentials. In the scenario without BogusBiter, most or perhaps all credentials collected by a phisher are real credentials submitted by victims, thus the *real-to-all* ratio is close to one. A phisher can easily verify these credentials at the legitimate website, assess their values, and ultimately use them to obtain money.

In the scenario of anti-phishing with BogusBiter (Figure 3.2), a phishing site receives both real credentials and bogus credentials. Real credentials came from cheated users, that is, users who visited the phishing site and meanwhile became victims by revealing their credentials. The ratio between the number of cheated users and the total number of phishing site visitors can be denoted as *cheat-to-visit*. This ratio is often used by researchers to estimate the severity of phishing attacks. So if the total number of phishing site visitors is N , the number of real credentials being collected at the phishing site becomes “ $N * \textit{cheat-to-visit}$.” Meanwhile, because BogusBiter submits a set of S credentials in each phishing site visit (either by a cheated user or by a security-conscious user as explained in the design overview), the total number of credentials being collected at the phishing site becomes “ $N * S$.” Therefore, in the scenario of anti-phishing with BogusBiter, the *real-to-all* ratio can be computed as: $\frac{\textit{cheat-to-visit}}{S}$.

If all the phishing site visitors become victims, the *cheat-to-visit* ratio equals one. Therefore, the upper bound of the *real-to-all* ratio is $\frac{1}{S}$. However, the experiments conducted

by Jakobsson and Ratkiewicz [47] demonstrate that even with the effects of modern anti-phishing efforts, about $11 \pm 3\%$ of users will read a spoofed email, visit the phishing site, and enter their login credentials. In addition, Garera et al. [35] found that on average, 8.24% of users become victims after visiting phishing sites. If we use 10% as a realistic value for the *cheat-to-visit* ratio, the *real-to-all* ratio becomes $\frac{1}{10S}$. Thus, if the value of the set size S is 10, a real credential will be hidden among 100 bogus credentials. Moreover, it is plausible to assume that the *cheat-to-visit* ratio will decrease in the long run due to technical advances and educational efforts — a trend that favors BogusBiter. Assuming that the indiscernibility requirement is achievable, we now analyze the probability and the expected number of tries for a phisher to single out a certain number of real credentials by verifying them at the legitimate website. Since each set of S credentials is submitted by BogusBiter from a user’s browser within a few milliseconds, a phisher can easily group the collected credentials by sets and verify them. If a set of S credentials is submitted from a victim’s browser, the real credential will be singled out by a phisher with an expected number of $\frac{S+1}{2}$ tries. However, because a phisher cannot discern which set includes a real credential, the phisher has to verify all sets of the collected credentials in order to single out as many real credentials as possible. Considering the very low *cheat-to-visit* ratio, without loss of generality, we simplify our analysis by mixing together all sets of the collected credentials. Let n be the total number of credentials collected at a phishing site, and m be the number of real credentials revealed by victims. Let X_k be the discrete random variable representing the number of tries performed by the phisher to single out k real credentials. Let $P_r(X_k = i)$ be the probability of “ $X_k = i$ ” and $E[X_k]$ be the expectation of X_k . Intuitively, $P_r(X_k = i)$ is the probability that a phisher identifies the k^{th} real credential until the i^{th} try. That is,

in the first $i - 1$ tries, the phisher has identified $k - 1$ real credentials; meanwhile, at the i^{th} try, the phisher also identifies a real credential. Therefore, based on the definition of the binomial coefficient, we can calculate $P_r(X_k = i)$ and $E[X_k]$ using Formula (3.1) and Formula (3.2), respectively, where $\sum_{i=k}^{n-m+k} P_r(X_k = i) = 1$ and $k = 1, 2, \dots, m$.

$$P_r(X_k = i) = \frac{\binom{n-m}{i-k} \binom{m}{k-1}}{\binom{n}{i-1}} \cdot \frac{m - (k - 1)}{n - (i - 1)} \quad (3.1)$$

$$\begin{aligned} E[X_k] &= \sum_{i=k}^{n-m+k} i \cdot P_r(X_k = i) \\ &= \sum_{i=k}^{n-m+k} i \cdot \frac{\binom{n-m}{i-k} \binom{m}{k-1}}{\binom{n}{i-1}} \cdot \frac{m - (k - 1)}{n - (i - 1)} \end{aligned} \quad (3.2)$$

With the *cheat-to-visit* ratio set to 10%, Figure 3.3(a) illustrates the expected number of tries for a phisher to single out one real credential, that is, $E[X_1]$. The four curves correspond to four different values of set size S . For example, if there are 6 real credentials hidden among all the collected credentials, to single out one real credential, the expected number of tries are 69 and 103, for set sizes 8 and 12, respectively. Figure 3.3(b) illustrates the expected number of tries for a phisher to single out all real credentials. Similarly, if there are 6 real credentials hidden among all the collected credentials, to single out these 6 real credentials, the expected number of tries are 412 and 618, for set sizes 8 and 12, respectively. From this example, we can see that a set size of 8 can already allow BogusBiter to feed a relatively large number of bogus credentials into a phishing site and well hide victims' real credentials among bogus credentials. However, we should note that such a hiding effect will never be enough to frustrate greedy phishers who intend to verify all the collected credentials. Therefore, a defensive line enabled by BogusBiter is highly

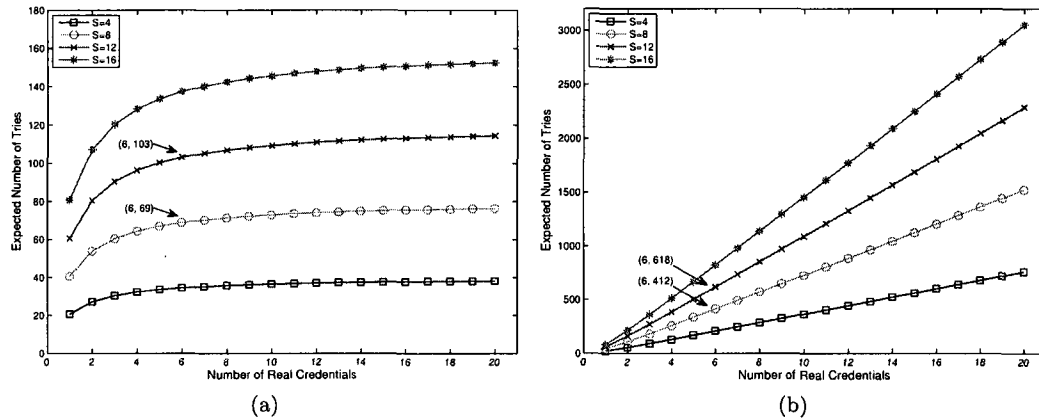


Figure 3.3: Expected number of tries for a phisher to single out: (a) one real credential, (b) all real credentials.

desirable (see Section 3.3.3).

3.3.2.2 Indiscernibility

The indiscernibility requirement is essential for BogusBiter to work. It has two implications:

- (1) the submission actions initiated from victims' browsers should be very difficult to be differentiated from the submission actions initiated from security-conscious users' browsers,
- and (2) victims' real credentials should be very difficult to be differentiated from bogus credentials generated by BogusBiter.

For a victim who ignores a browser's phishing warning, BogusBiter first intercepts the credential submission HTTP request before it is sent out. Next, BogusBiter creates $S - 1$ bogus credentials based on the victim's real credential and spawns $S - 1$ new HTTP requests based on the original HTTP request. Each of the $S - 1$ spawned requests is exactly the same as the original request, except for carrying a bogus credential instead of a real one. Then, BogusBiter inserts the original HTTP request into the $S - 1$ spawned requests and

sends out all the S requests within a few milliseconds. Finally, BogusBiter interprets and properly processes the returned HTTP responses so that a phishing site cannot identify the differences between the S submissions.

For a security-conscious user who accepts a browser's phishing warning, BogusBiter first imitates a victim's behavior by entering a generated bogus credential into the phishing page and submitting it. Next, similar to the above case for a real victim, BogusBiter intercepts this original HTTP request, spawns $S - 1$ new HTTP requests, and generates the corresponding $S - 1$ bogus credentials as well. Finally, BogusBiter sends out the S requests and processes the returned responses in the same way as it does for a victim, thereby making it hard for a phisher to distinguish these submissions from those initiated from a victim's browser.

As for bogus credential generation, BogusBiter uses the original credential as the template to generate the $S - 1$ bogus credentials. For a victim, the original credential is the victim's real credential and thus is ready to use. For a security-conscious user, the automatically generated original credential should be similar to a human's real credential. In current design, BogusBiter randomly generates a username/password pair as the original credential. For the remaining $S - 1$ bogus credentials, a specific rule should be followed to generate them so that neither a human nor a computer can easily discern which is the original credential and which are the rest. We will present the rule used by BogusBiter in Section 3.3.3.

3.3.2.3 Usability

In terms of usability, the major advantage of BogusBiter is its transparency to users. Complementary to existing preventive anti-phishing approaches, BogusBiter automatically defends against phishing attacks without user involvement. Meanwhile, because BogusBiter only needs to submit some extra bogus credentials to a suspected phishing site and does not contact any third-party service, it will not cause any security or privacy problems.

The main usability concerns come from the scenario of a false positive (i.e., a legitimate website is wrongly classified as a phishing site). While the occurrence of false positives is rare for Firefox 2, IE 7, and recent detection techniques as mentioned in Section 3.2, BogusBiter should eliminate or reduce the possible side effects on users' access to misclassified legitimate websites.

The first side effect is that submitting a set of S login requests and waiting for responses will induce an additional delay to users. To reduce the delay, BogusBiter sends out all the S requests within a few milliseconds, so that the round-trip times of the S submissions can be overlapped as much as possible. Accordingly, as long as the set size S is not too large, the additional delay incurred by BogusBiter should be minimal and unobtrusive. Our experimental results in Section 3.5 confirm that the additional delays are negligible.

The second side effect is that a user's real account may be locked because multiple login requests are submitted from the user's browser to a legitimate website within a few milliseconds. To defend against password guessing attacks, some websites may lock a user's account for a period of time after several failed login attempts. However, because all the usernames are different for the S login requests sent out by BogusBiter, the "account with

many failed login attempts” alarm will not be triggered as discussed in [85]. Our experiments on 20 legitimate websites confirm that account locking is not a concern for BogusBiter.

The third side effect is that a user may be asked to complete a CAPTCHA [3] test, for the same reason that multiple login requests are submitted from the user’s browser within a few milliseconds. Some websites may resort to this mechanism to counter password guessing attacks or denial of service attacks. However, in our legitimate site experiments where false positives are assumed to occur, no CAPTCHA test is triggered if the set size S is not greater than 10, and only two of the 20 websites ask a user to do a CAPTCHA test if the set size S is greater than 10.

3.3.3 Defensive Line

Simply requiring BogusBiter to meet the offensive objective is not sufficient. This is because even if victims’ real credentials are well hidden among bogus credentials, a phisher can still visit the legitimate website to verify each of the collected credentials. Therefore, a defensive line is highly desirable, and BogusBiter should enable a legitimate website to exploit the verification process initiated (either manually or automatically) by the phisher to detect victims’ stolen credentials in a timely manner.

3.3.3.1 Working Mechanism

BogusBiter makes such a defensive line feasible by imposing a *correlation requirement* upon the generation of the $S - 1$ bogus credentials. It is important that this correlation requirement should not violate the indiscernibility requirement of credential generation, that is, victims’ real credentials should be very difficult to be differentiated from bogus

credentials generated by BogusBiter.

- ***Correlation Requirement:*** Based on the original credential, a specific rule is applied to generate the $S - 1$ bogus credentials. This rule must guarantee that the S credentials in a set are correlated: given any one of them, we can reversely derive a small superset that includes all the S credentials.

BogusBiter attempts to meet both the correlation and indiscernibility requirements on credential generation by using a simple *substitution rule*. While there are other ways to meet the two requirements, we choose the substitution rule because of its simplicity and efficiency for verification. Due to our empirical experience that if the set size S is not greater than 10, no usability problem occurs and the delay overhead is small (see Section 3.5), the substitution rule is tailored to have $S \leq 10$. Note that the exact value of S should be publicly known.

To generate the $S - 1$ bogus username/password pairs, BogusBiter first computes an integer position i between 1 and S inclusively. This integer position determines which set of $S - 1$ bogus credentials will be generated, and it also determines in which order the S credentials will be sent out to a phishing site. BogusBiter uses Formula (3.3) to deterministically compute this integer position i :

$$i = PRF(k, original_username) \bmod S + 1, \quad (3.3)$$

where k is a master secret that is randomly chosen when a BogusBiter is installed or configured, and PRF is a secure pseudorandom function. The master secret k is securely stored and used by BogusBiter. A user does not need to memorize the master secret, but is allowed to export and use the same master secret on different computers. From a given original credential, the same $S - 1$ bogus credentials will always be generated, and the S credentials will always be submitted to a phishing site in the same order. Therefore, even if a phisher can attack a victim multiple times, the phisher cannot find the real credential by observing which credential over time appears most often. Meanwhile, since this formula only securely hashes the original username, it is applicable both to websites that ask a user to submit username/password pair at the same time, and to websites that require a user to first submit a username and then submit a password.

Next, BogusBiter identifies the first digit in the original username as the username replacement character, denoted as *username-rc*; if the original username does not contain a digit, the first letter (upper or lower case) is identified as the username-rc. Using the same method, BogusBiter identifies the password replacement character in the original password, denoted as *password-rc*.

Then, for each integer position j from 1 to S inclusively where $j \neq i$, BogusBiter generates a bogus username/password pair by substituting both the username-rc character and the password-rc character in the original username/password pair using one of the following two replacement methods:

- (1) For the case of $j - i > 0$: if username-rc (also for password-rc) is a letter, this lower (or upper) case letter is replaced by another lower (or upper) case letter $j - i$ places further down the alphabet, wrapped around if needed, i.e., 'z' is followed by

‘a’ (or ‘Z’ is followed by ‘A’); if username-rc (also for password-rc) is a digit, this digit is replaced by another digit $j - i$ places further down the single digit sequence “0123456789”, wrapped around if needed, i.e., ‘9’ is followed by ‘0’.

- (2) For the case of $j - i < 0$: if username-rc (also for password-rc) is a letter, this lower (or upper) case letter is replaced by another lower (or upper) case letter $i - j$ places further up the alphabet, wrapped around if needed, i.e., ‘a’ is followed by ‘z’ (or ‘A’ is followed by ‘Z’); if username-rc (also for password-rc) is a digit, this digit is replaced by another digit $i - j$ places further up the single digit sequence “0123456789”, wrapped around if needed, i.e., ‘0’ is followed by ‘9’.

Position	Username/Password
j=1	(kcsmith/Fuzzycat95)
j=2	(lcsmith/Fuzzycat05)
→i=3	(mcsmith/Fuzzycat15)
j=4	(ncsmith/Fuzzycat25)

Table 3.1: Substitution from the original username/password pair (*mcsmith/Fuzzycat15*).

Username/Password
(icsmith/Fuzzycat75)
(jcsmith/Fuzzycat85)
(kcsmith/Fuzzycat95)
→(lcsmith/Fuzzycat05)
(mcsmith/Fuzzycat15)
(ncsmith/Fuzzycat25)
(ocsmith/Fuzzycat35)

Table 3.2: Derivation from the username/password pair (*lcsmith/Fuzzycat05*).

Table 3.1 illustrates an example of applying the substitution rule to the original username/password pair (*mcsmith / Fuzzycat15*). In this example, the username replacement character username-rc is the first ‘m’ in the original username and the password replacement character password-rc is the digit ‘1’ in the original password. These two alphanumeric characters will be replaced to generate $S - 1$ bogus credentials. If $S = 4$ and the computed integer position i is 3, three bogus username/password pairs are generated for $j=1, 2$, and 4, respectively.

```

Algorithm: SCI (f-uname/f-pword)
1. Initialize the result set as empty :  $R = \emptyset$ ;
2. Construct the set :  $D = \{(d\text{-uname}/d\text{-pword}) : (d\text{-uname}/d\text{-pword})$ 
   is a credential derived from  $(f\text{-uname}, f\text{-pword})\}$ ;
3. for each  $(d\text{-uname}/d\text{-pword}) \in D$  do
4.   if  $d\text{-uname}$  matches a valid account's username then
5.     if  $d\text{-pword}$  matches the valid account's password then
6.        $R = R \cup \{(d\text{-uname}/d\text{-pword})\}$ ;
7.     endif
8.   endif
9. endfor
10. return the result set  $R$ ;

```

Figure 3.4: The Stolen Credential Identification (SCI) procedure.

Finally, BogusBiter submits the S username/password pairs to a suspected phishing site following their corresponding position order. Using Formula (3.3) to compute the integer position i and using their position order to send out the S credentials, BogusBiter makes it hard for a phisher to narrow down a victim's real credential even if the victim visits a phishing site twice from the same browser and enters the real credential twice. However, we should note that the overall extent to which the indiscernibility requirement can be met still depends on the characteristics (such as meaningfulness) of a victim's real credential. We further discuss this limitation in Section 3.6.3.1.

Clearly the substitution rule above meets the correlation requirement. Given any one of the S credentials, we can derive at most $2 * (S - 1)$ variations based on the substitution rule, in which further down replacement produces $S - 1$ variations and further up replacement produces other $S - 1$ variations. These $2 * (S - 1) + 1$ credentials cover all the S credentials submitted to the phishing site. Table 3.2 lists an example derivation from the credential (lcsmith / Fuzzycat05).

Now let us see how a legitimate website can take advantage of the correlation requirement to identify the credentials stolen by phishing attacks. If a phisher is lucky enough (with $\frac{1}{S}$ probability) to choose a victim's real credential as the first try to verify at the legitimate website, this login attempt will succeed and the legitimate website cannot detect the fact that a real credential has been stolen and verified. However, for any failed login attempt, the legitimate website will trigger the procedure of Stolen Credential Identification (SCI), which is illustrated in Figure 3.4. SCI takes the failed username/password pair (`f-username/f-password`) as its input. It constructs the set D of derived credentials (line 2), and seeks a match between a derived username/password pair and a valid account's username/password pair. Then, it adds any derived username/password pair (`d-username/d-password`) that matches a valid account's username/password pair to the result set R (line 6). SCI finally returns the result set R as its output.

If the failure of a login attempt is caused by a phisher who is verifying any one of the $S-1$ bogus credentials generated from a victim's real credential, SCI must report a match since the derived credential set D contains the victim's real credential. The matched credential is the victim's real credential that has been revealed to the phisher, and is included in the result set R . However, if the failure of a login attempt is due to any other reasons, even if there is a chance that a derived username `d-username` may match a valid account's username (line 4), the probability that the correspondingly derived password `d-password` also happens to match this valid account's password (line 5) is extremely low. This probability is equivalent to that of randomly guessing a valid account's password. As an example, if a user accidentally mistypes the user's real password (or an attacker launches online password guessing attacks against a user), the login attempts will fail but SCI will not report a match.

Therefore, if the result set R is not empty, the username/password pair (the probability of having two or more credential pairs in the result set R is also extremely low) contained in R must have been stolen by a phisher. The legitimate website can take immediate actions to protect the victim even before the phisher figures out the victim's real credential. Because SCI is turned on only when a login attempt fails and it only needs a small number of verifications (at most $2 * (S - 1)$ for our substitution rule), the overhead is very small for a legitimate website. If necessary, this identification task can even be delegated to a separate machine.

3.3.3.2 Deployment of Defensive Line

While BogusBiter is installed in a user's Web browser, the defensive line enabled by BogusBiter needs to be deployed only on those legitimate websites that are really targeted by phishers. These phishing-targeted legitimate websites listed in the APWG database [138] usually have properly registered domain names and well-designed webpages, and may even be whitelisted by some phishing detection tools. None of their login pages will be misclassified as phishing pages by popular detection tools. The rare false positives [131, 151] produced by phishing detection tools are mainly caused by some legitimate websites that are almost never targeted by phishing attacks. We do not need to deploy the defensive line of BogusBiter on them.

Moreover, the deployment work on phishing-targeted legitimate websites is very simple because SCI only uses these websites' existing authentication information and does not change their authentication mechanisms (no matter plaintext-equivalent mechanisms or verifier-based mechanisms). This server-side deployment cost is minimal compared to that

of Dynamic Security Skins [19], which changes authentication mechanisms via the SRP protocol [116], and to that of BeamAuth [1], which demands an extra secret token for every user account.

3.3.3.3 Scale-Independency Properties

The defensive line enabled by BogusBiter also has two valuable scale-independency properties. First, the efficacy of the defensive line does not depend on the *cheat-to-visit* ratio, that is, it does not require a large percentage of users to properly respond to anti-phishing warnings. Second, the efficacy does not depend upon a massive installation of BogusBiter in users' browsers, i.e., even a single vulnerable user who installs BogusBiter can benefit from a deployed defensive line. These two scale-independency properties are not only valuable by themselves, they also ensure that BogusBiter cannot be easily evaded by sophisticated phishers, as will be discussed in Section 3.6.

3.4 Implementation

We have implemented BogusBiter as a Firefox extension in approximately 1700 lines of JavaScript code and 100 lines of C++ code. Seamlessly integrated with the built-in phishing protection feature of Firefox 2 [148], BogusBiter consists of four main modules: Information Extraction, Bogus Credential Generation, Request Submission, and Response Process, as illustrated in Figure 5.2. We detail these four modules in the remainder of this section.

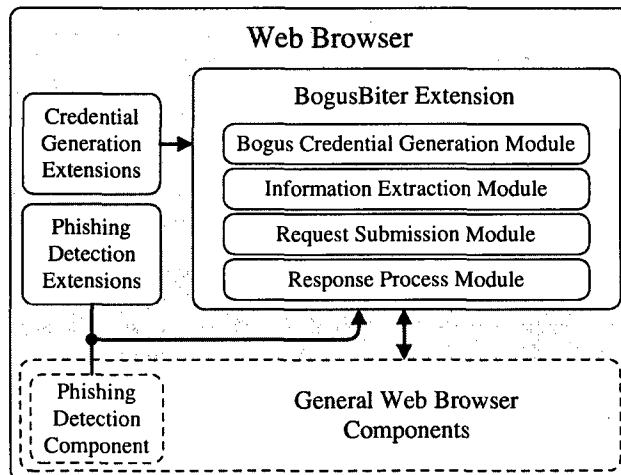


Figure 3.5: Implementation of BogusBiter as a Firefox 2 browser extension.

3.4.1 Information Extraction Module

The information extraction module extracts the username and password pair and its corresponding form element on a login page by analyzing Document Object Model (DOM) objects. First, all the `HTMLInputElement` objects within the `HTMLDocument` object of the login page are collected. Next, the password object is located by examining its special attribute `type="password."` A similar password locating method is also used in [31, 91]. Then, the `HTMLFormElement` object—the submission form object—associated with the password object is directly extracted. Finally, based on object attributes, the username object is extracted from other `HTMLInputElement` objects included in the submission form. Following this element extraction order combined with the attribute analysis of the visible input fields, BogusBiter can accurately identify username, password, and form elements on a login page. Note that phishers may use non-standard login pages to disrupt this automatic information extraction procedure and evade BogusBiter. We discuss some solutions to this kind of evasions in Section 3.6.3.2.

The information extraction module also implements a protection mechanism to defend against *input-stealing* attacks that use malicious JavaScript code on a phishing page to directly steal a victim’s credential. Existing works such as AntiPhish [56] and PwdHash [91] provide good technical guidance for implementing such a protection mechanism. In BogusBiter, we adopt the *keystroke intercepting* technique introduced in PwdHash and create protectors on username and password elements. More precisely, username and password keystrokes are intercepted by the registered event handlers and are masked to hide from the JavaScript on a webpage. Therefore, a victim’s real keystrokes are recorded by BogusBiter, but are blocked from being received by various JavaScript attacks [91]. We choose the *keystroke intercepting* technique because it is more generic than the *temporary deactivating* technique used in AntiPhish’s Firefox version.

3.4.2 Bogus Credential Generation Module

The bogus credential generation module generates $S - 1$ bogus credentials based on an original credential. For a victim, the original credential is the victim’s real credential. For a security-conscious user, in our current implementation, this module will randomly generate a username/password pair composed of upper/lower case letters and digits as the original credential. Advanced original credential generation methods can also be incorporated into BogusBiter, so that a randomly generated original credential will look more like a real user’s credential. The substitution rule of BogusBiter is implemented in JavaScript. The open source HMAC_SHA256_MAC() JavaScript function implemented by Poettering [164] is used as the secure pseudo-random function of Formula (3.3).

3.4.3 Request Submission Module

The request submission module is responsible for spawning and submitting multiple HTTP requests. Its implementation is guided by both the indiscernibility and usability requirements of BogusBiter. Since each HTTP request can only carry one credential, S requests are needed to submit a set of S credentials to a phishing site. For a victim, once a credential is entered and the submit button is clicked, the first HTTP request is initiated from the current browser window. For a security-conscious user, the action of accepting a phishing warning triggers BogusBiter to imitate a human's credential entering and button clicking actions and initiate the first HTTP request from the current browser window.

Next, just *before* the first HTTP request is actually sent out, BogusBiter is notified by Firefox's global notification service and intercepts this HTTP request. Then, BogusBiter quickly spawns the other $S-1$ HTTP requests with each of them carrying a bogus credential. The main challenge here lies in how to efficiently spawn $S-1$ new HTTP requests and schedule the submission of all the S requests. A few solutions are available, for example, using multiple submission windows, or reusing the submission form on one browser window to submit multiple times. However, they suffer from various usability drawbacks such as multiple webpage refreshing and long interaction time.

BogusBiter, instead, creates and uses internal HTTP channels to submit requests behind the screen. In order to make our extension code more portable, we choose to use XMLHttpRequest objects [192] to create internal HTTP channels. XMLHttpRequest objects are supported by both Firefox 2 and IE 7, and they allow JavaScript to perform HTTP client functionalities such as submitting form data or loading data from a server. The first HTTP

request is also associated with an HTTP channel, which is created by the browser. For this first HTTP request, all its contents, such as message header and message body [27] can be extracted from its HTTP channel. Then, $S - 1$ XMLHttpRequest objects are created and their corresponding HTTP channels are established based on the contents extracted from the first HTTP channel. More specifically, BogusBiter executes the following four steps: request initialization, message body replacement, header fields setting, and header fields reordering:

- (1) *Request Initialization:* For each of the $S - 1$ XMLHttpRequests, the same request type and URL as those in the first HTTP request are used. Asynchronous mode is used so that request sending is nonblocked and all the corresponding HTTP responses can be handled in a specified callback function. Since HTML forms, especially login forms, are in general submitted using POST instead of GET type of HTTP requests for security reasons, we only consider POST type of HTTP requests in the following discussion. Indeed, it is much simpler to process the GET type of HTTP requests.
- (2) *Message Body Replacement:* For each of the $S - 1$ XMLHttpRequests, BogusBiter only needs to make a copy of the message body extracted from the first HTTP request, and then replace the original username/password pair with a bogus username/password pair. Nothing else needs to be changed in the message body. Because the bogus username/password pair and original username/password pair have the same length, the message body length does not change. Meanwhile, since the first HTTP request's message body is extracted before its HTTP channel is encrypted, this message body replacement also works correctly for secured (HTTPS) connections.

- (3) *Header Fields Setting:* For each of the $S - 1$ XMLHttpRequests, the “Content-Type” request header field is set as “application/x-www-form-urlencoded” to mimic the case of submitting a form on a browser window. The “Content-Length” request header field is set to the same value as that of the first HTTP request, because the message body length is unchanged. The “Referer” request header field also needs to be set as the same value as that of the first HTTP request. The “Cookie” request header field is automatically set by the Firefox Web browser because the same URL has been specified. For each of the $S - 1$ XMLHttpRequests, BogusBiter also makes sure that “no-cache” is assigned to both the “Pragma” request header field and the “Cache-Control” request header field so that the form submissions will not be cached, and the “close” is assigned to the “Connection” request header field so that its TCP connection will not be persisted and shared with any other requests. The first HTTP request also needs to be adjusted to have the same values for these three request header fields.
- (4) *Header Fields Reordering:* For each of the $S - 1$ XMLHttpRequests, the request header fields must be reordered so that the same order used in the first HTTP request will be used. Since the order of request header fields is not significant as defined in [27], this reordering will not cause any problem. In our implementation, only the order of “Content-Type,” “Content-Length,” “Pragma,” “Cache-Control,” “Cookie,” and “Referer” is adjusted by BogusBiter, due to some subtle implementation differences between an XMLHttpRequest and a regular HTTP request in Firefox. To support this reordering, we actually introduced a new function `switchHeaderField-`

sPosition(headerFieldA, headerFieldB) to Firefox's nsIXMLHttpRequest interface and nsIHttpChannel interface and implemented this new function in C++. This new function may also be useful for other applications that use XMLHttpRequest objects.

After the completion of these four steps, the $S - 1$ XMLHttpRequests and the first HTTP request all have the same request type, URL, header fields, and header field order. Their message bodies are all the same except for carrying different username/password pairs. As previously described in BogusBiter's substitution rule, the submission order of these S requests is decided when the $S - 1$ bogus credentials are generated. If the i th position ($1 \leq i \leq S$) is computed for the original credential, BogusBiter first asynchronously transmits $i - 1$ XMLHttpRequests, which carry the first $i - 1$ bogus credentials. Then, BogusBiter transmits the first initiated HTTP request, which carries the original credential. Finally, BogusBiter asynchronously transmits the remaining $S - i$ XMLHttpRequests, which carry the last $S - i$ bogus credentials. All the S requests are sent out within a few milliseconds, and no timing clue can be observed on a Web server or proxy.

3.4.4 Response Process Module

After receiving and interpreting an HTTP request, a website replies with an HTTP response message. For a legitimate website, if the credential carried in a request is valid, a successful login page is returned in the response message; otherwise, a failed login page is returned in the response message. Phishing sites may take different response actions after receiving credential submission requests (see Section 3.5.2). In every case, BogusBiter parses and renders the response message of the first HTTP request on the browser window, and processes the response messages of the $S - 1$ XMLHttpRequests behind the screen using

a callback function. Therefore, BogusBiter can always correctly match responses to their corresponding requests and work transparently to users.

Many times there are Web objects such as JavaScript and embedded images associated with each response message. These objects will be downloaded by the Web browser if the response message corresponds to an HTTP request initiated from the browser window, but by default will not be downloaded if the response message corresponds to an XMLHttpRequest. Future phishers may want to discern which are XMLHttpRequests by exploiting this fact and manipulating response contents. For example, a phisher may return a different HTML page to each submission, which includes a slightly different named image. Later on, by examining whether an image has ever been downloaded from the phishing site, the phisher can identify bogus credentials submitted by XMLHttpRequests.

To defend against such *rendering-based* attacks, BogusBiter utilizes a set of *hidden DOM windows* to render these asynchronously returned response pages for XMLHttpRequests, thus leaving no clue to phishers. Because the same Web objects cached by a browser will be directly used by different DOM windows, bandwidth-overhead incurred by this mechanism is negligible, especially for legitimate websites. For a phishing site that returns a different HTML page for each of BogusBiter's S submissions, the possible long delay due to downloading different Web objects will only annoy a victim and encourage the victim to leave the phishing site—a result that actually favors victims' interests.

3.5 Evaluation

We conducted three sets of experiments to evaluate BogusBiter. In the first set of experiments, we built a testbed to verify the implementation correctness of BogusBiter with respect to indiscernibility. In the second and third sets of experiments, we ran BogusBiter against 50 phishing sites and 20 legitimate websites to validate its efficacy, in terms of attacking capability and usability.

3.5.1 Testbed Experiments

In the testbed experiments, we set up an Apache 2 Web server in a Linux machine and hosted over twenty various phishing webpages on it. We used BogusBiter to send various login requests to these phishing webpages either directly or through proxies. By examining both request logs and request contents at the Web server, we verified that all the S requests in a set are exactly the same, except for the credentials carried in the request bodies. In addition, we placed an open-source tool, Tcpmon [185], in between the Web browser and Web server to monitor TCP connections. We verified that the S submission requests are transmitted over S independent non-persistent TCP connections; therefore, it is hard for a phisher to differentiate these requests at the TCP connection level.

3.5.2 Phishing Site Experiments

In the phishing site experiments, we ran BogusBiter against 50 verified phishing sites chosen from PhishTank [176]. PhishTank is a community based anti-phishing service and its data have been widely used for evaluating phishing detection techniques [68, 131, 132, 146]. These 50 chosen phishing sites are diverse in terms of their locations, design styles, and targeted

brand names. For each phishing site, when it was online, we tested BogusBiter with four different set sizes of 4, 8, 12, and 16. Our major experiential findings are summarized as follows.

First, BogusBiter is capable of attacking all the 50 phishing sites. Acting as either a victim or a security-conscious user, BogusBiter always works correctly: it sends out all the S requests within 10 milliseconds, and then processes all the responses properly. In rare cases that phishing sites were not correctly detected by Firefox 2, we manually corrected the detection results to trigger BogusBiter.

Second, the delay caused by BogusBiter is minimal when the set size S is 4 or 8. Here the delay means the submission interaction time difference between using BogusBiter and not using BogusBiter. The submission interaction time is the time elapsed between the transmission of the first request and the reception of the last response. Figure 3.6(a) depicts the percentage of phishing sites versus the delay caused by BogusBiter under four different set sizes. We can see that if the set size S is 4 or 8, for over 85% of phishing sites, the delay is less than 4 seconds. This delay measure is common to either a security-conscious user or a victim, but the delay effect is different. A security-conscious user is unaware of such a delay because the user is actually redirected to a default webpage by Firefox. A victim may perceive this delay because the victim is waiting for the response from the phishing site. Nevertheless, it is definitely worthwhile adding a small delay on revealing a victim's credential, in order to make it less likely for phishers to succeed.

Third, phishing sites take three different response actions after receiving a user's credential submission request. Among 50 phishing sites, 38 of them simply redirect a user to the invalid login pages of the targeted legitimate websites; 11 of them keep a user at their

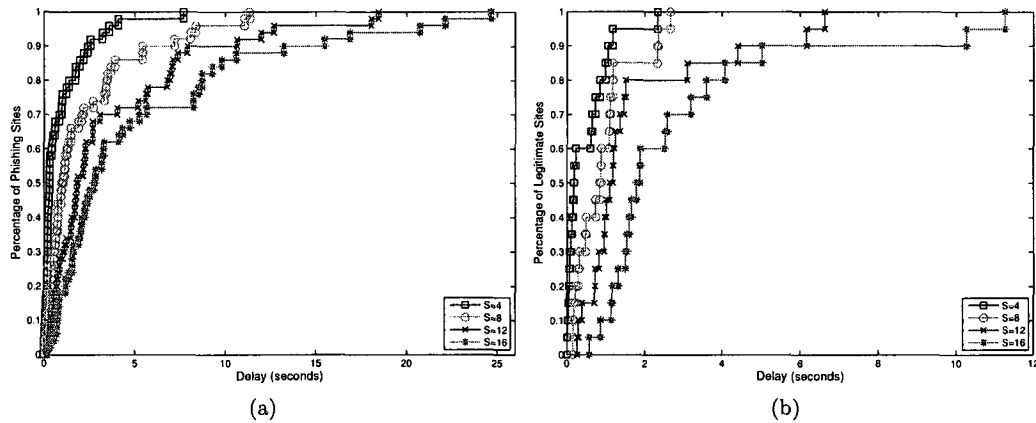


Figure 3.6: Delay caused by BogusBiter on: (a) phishing sites, (b) legitimate sites, under different set size S .

local sites by using more faked webpages; and the last phishing site is very tricky because it verifies the received credential in real time at the legitimate website and then sends back a response based on the verification result. If a user submits a valid credential, the phishing site steals the credential and then redirects the user to the legitimate website; otherwise, it lets the user re-login on the phishing site. All three types of response actions attempt to continue deceiving a victim and prevent the victim from realizing that an attack has happened. But the third type of response action not only obtains and verifies a credential in real time, it is also more deceptive to vulnerable users. The defensive line of BogusBiter indeed provides an excellent opportunity for a legitimate website to defend against such attacks in real time.

3.5.3 Legitimate Site Experiments

In the legitimate site experiments, we ran BogusBiter against 20 legitimate websites listed in Table 3.3. None of these websites is classified as a phishing site by either Firefox 2 or

paypal.com	amazon.com	gmail.com	cox.com	myspace.com
ebay.com	buy.com	yahoo.com	sprint.com	walmart.com
citibank.com	ecost.com	msn.com	geico.com	careerbuilder.com
53.com	ubid.com	aol.com	aaa.com	my.wm.edu

Table 3.3: The 20 legitimate websites.

IE 7. We intentionally set the detection results as phishing to simulate false positive cases, and used real accounts on these legitimate websites to evaluate the usability of BogusBiter. We summarize the major experimental results as follows.

First, as we expected, none of these legitimate websites lock a real account during our extensive tests. Second, if the set size S is 4 or 8, none of these legitimate websites require CAPTCHA tests. If the set size S is 12 or 16, only two websites ask a user to do a CAPTCHA test after receiving S credentials. This test is a burden to a user but will not block a user's further interactions with a Web server. Third, the delay caused by BogusBiter is very small when the set size S is 4 or 8. Figure 3.6(b) depicts the percentage of legitimate sites versus the delay caused by BogusBiter under four different set sizes. We can see that if the set size S is 4 or 8, for all the 20 legitimate sites the delay is less than 3 seconds, and for over 85% of legitimate sites the delay is less than one second. Therefore, BogusBiter only induces a very small delay to users even if false positives really occur. The delay on legitimate websites is much smaller than that on phishing sites, since the request processing capability of legitimate websites is generally higher than that of phishing sites.

3.6 Discussions

In this section, we discuss the deployment scale of BogusBiter, the preparations that may be needed for BogusBiter's massive deployment, and the limitations of BogusBiter.

3.6.1 Deployment Scale

As discussed in Section 3.3.3.2, the defensive line (the SCI procedure) enabled by BogusBiter needs to be deployed only on those legitimate websites that are really targeted by phishers. So, here we only discuss the deployment of the BogusBiter browser extension. Like most client-side protection mechanisms, BogusBiter protects only those users who install it. On one hand, due to its *scale-independency* properties, the defensive line enabled by BogusBiter can effectively identify the stolen credentials whose owners use BogusBiter, no matter how many users install BogusBiter and what percentage of them are real victims. On the other hand, the power of BogusBiter's offensive line against a phishing site is scaled to the number of users who install BogusBiter. With the increase of BogusBiter users, victims' real credentials can be better hidden among bogus credentials. Therefore, in order to protect as many users as possible, BogusBiter should be deployed as widely as possible. Ideally, if BogusBiter could be integrated into popular Web browsers as a built-in feature, a ubiquitous deployment will be easily achieved and the benefits brought by BogusBiter will be maximized.

3.6.2 Massive Deployment Preparation

When BogusBiter is integrated into popular Web browsers, it can be triggered with high confidence for blacklisted phishing login pages; it can also be triggered for suspicious (but not blacklisted) phishing login pages hosted on less popular websites. The main concern about such a massive deployment of BogusBiter is that if the login page of a legitimate site is wrongly flagged as a phishing page, the load on the site's authentication servers will increase by a factor of S due to BogusBiter. However, the false positives produced by

widely deployed phishing detection mechanisms such as used in IE 7 and Firefox 2 are rare, especially for popular websites that have a large number of users. This is because otherwise the false positives would have been noticed and corrected by these websites to prevent losing users. As reported in [131], both IE 7 and Firefox 2 achieve a zero false positive rate for 516 representative legitimate websites. Thus, we expect that only few less popular and poorly designed legitimate websites need to prepare for a massive deployment of BogusBiter.

We suggest two simple solutions for these websites to prepare. Let us assume that BogusBiter's functionality is integrated into a new version of IE or Firefox Web browser. Using the browser, the operator of a legitimate website can easily verify whether the site's login page will be incorrectly classified as a phishing page. If a misclassification does occur, two simple solutions exist. One solution is to report this misclassification and request the Web browser vendor to either remove this legitimate site from the blacklist or add it to the whitelist. The other solution is to revise the login page of this site, for example by removing suspicious features, so that the page can pass heuristic-based tests [17, 35, 68, 132]. We suggest these preparations not merely for the need of BogusBiter's massive deployment. Indeed, legitimate websites may lose customers if they do not take active measures to reduce their chances of being misclassified.

3.6.3 Limitations of BogusBiter

Should BogusBiter become widely deployed, phishers may explore its limitations to circumvent it. In general, the potential evasions can be divided into offline evasions and online evasions.

3.6.3.1 Offline Evasions

In offline evasions, phishers analyze their collected credentials by using local username filtering techniques, meaningful credential filtering techniques, or statistical filtering techniques.

(1) *Local username filtering:* In BogusBiter’s design, we assume that a phisher does not have a complete list of valid usernames for a targeted legitimate website, and cannot directly query a targeted legitimate website for the validity of a specific username. Otherwise, a phisher can simply conduct local username filtering without doing remote credential verification. Currently, this assumption may not be valid for some websites. For example, Bank of America’s website can tell a user whether a login is valid before a password is entered. For these websites, we recommend them to hide their username validity information by using some protection methods such as suggested in [12] and [32], thus not just to receive better protection from BogusBiter, but also to provide a necessary defense against privacy leaking, invasive advertising and phishing, password guessing, and even DoS attacks [12, 32].

(2) *Meaningful credential filtering:* Using current substitution rule, BogusBiter may generate meaningless bogus credentials from users’ meaningful credentials (e.g., credentials containing dictionary words or human names), especially if an original username or original password does not contain a digit. Thus, a phisher may only select meaningful credentials to verify, while discarding the rest. Although this kind of meaningful credential filtering is error-prone because a victim’s real credential may be indeed meaningless and thus may be directly thrown away by a phisher, it can still be used by phishers to evade BogusBiter. Perhaps this is less of a concern for passwords, because the insecurity of low-entropy and

guessable passwords has long been recognized [40, 59, 75, 78], and more and more high security websites require users to choose passwords that contain at least one letter and one number.

(3) *Statistical filtering*: A phisher may also analyze the variations of credentials and use statistical language models such as bigrams or trigrams to identify victims' real credentials. However, we argue that this type of statistical filtering is also error-prone for the same reasons as already mentioned in the meaningful credential filtering. Unfortunately, we cannot obtain representative credential datasets to further analyze and support this argument. In addition, we need to emphasize that other new rules (in addition to our substitution rule) could also be designed to generate bogus credentials from an original credential. Especially, if those new rules take into account the statistical characteristics of credentials in representative datasets, they could better hide victims' real credentials among generated bogus credentials. It is worthy to design and apply such kinds of new rules, even if they may incur the cost of increasing the derivable credentials (see Table 3.2).

3.6.3.2 Online Evasions

Unlike offline evasions, in online evasions, phishers have to redesign their phishing sites and use special techniques to identify, in real time, which are real credentials submitted by victims. However, some inherent drawbacks limit the application and effectiveness of online evasion techniques. We now examine three representative classes of potential online evasion techniques.

(1) *JavaScript attacks*: A phisher may use two basic forms of JavaScript attacks to evade BogusBiter. One is an *input-stealing* attack that steals a user's credential using techniques

such as keystroke monitoring, and then sends back the results to the phishing site at form submission time or in real time. The other is a *rendering-based* attack that manipulates response contents to discern which are bogus credentials submitted by XMLHttpRequests. As discussed in Section 3.4.1 and Section 3.4.4, BogusBiter defends against these two basic forms of JavaScript attacks by using the *keystroke intercepting* technique and the *hidden DOM windows* technique, respectively.

More sophisticated JavaScript attacks can be launched by phishers. For example, a phisher can first have the phishing site code pause for a second or two to wait for BogusBiter submitting all the S credentials. The phisher can then present all of the S credentials back to a user, along with lines of a message “To improve our security process and defend your account against automated attacks, please select your username/password from this list of credentials.” If a user is fooled by such an attack, the phisher obtains the user’s credential. However, such attacks contain obvious hallmarks to distinguish themselves as malicious attacks that are specially fabricated to evade BogusBiter. Therefore, filtering functionalities can be added to BogusBiter to confidently detect and disable malicious JavaScript code. Note that detecting and filtering of malicious HTML content and JavaScript code is both desirable and feasible, and generic solutions can be found in recent research work such as SpyProxy [80] and BrowserShield [88].

(2) *Nonstandard login page*: A phisher may use nonstandard login pages to evade BogusBiter. A phisher may use a login form without the *type=“password”* HTML attribute, may write the entire phishing page in Flash, and may even display a virtual keyboard to users. For legitimate websites, using nonstandard login pages is not popular because it may cause some problems. For example, non-HTML login forms may create accessibility and

usability problems [112], and virtual keyboards are inconvenient to users and increase the risk of *shoulder surfing* attacks [152, 144]. Meanwhile, for phishing sites, using non-HTML login forms is also not popular because it makes a phishing attack more evident to users or phishing detection tools if its surface-level or deep-level characteristics become deviated from that of the targeted legitimate website. For these reasons, standard HTML pages remain the central focus of most anti-phishing research work [17, 56, 91, 115, 132].

Indeed, BogusBiter can borrow some solutions proposed by other researchers to defend against these attacks. For example, one solution can be borrowed from Dynamic Security Skins [19]. More specifically, a customized “trusted window in the browser dedicated to username and password entry” [19] can also be used by BogusBiter. A user is required to copy the entered username and password from the trusted window and paste them to the user recognized username and password fields in a login form. Using this solution, BogusBiter can intercept an original credential before filling a phishing login form. Since BogusBiter only needs to use a user’s paste actions to more accurately determine which are username and password fields, it can just paste a replaced bogus credential into a phishing login form and then do further replacements and submissions behind the screen. It is important to note that for BogusBiter, such a “trusted window” only needs to be triggered when a login page is classified as a phishing page and its username and password fields cannot be confidently identified. Also in such a case, BogusBiter will not be transparent to security-conscious users. After a security-conscious user clicks the “Get me out of here!” link on the phishing warning page, the user will be provided with the option to either really leave the site, or use the “trusted window” to help battle phishers by filling a bogus credential and identifying the username and password fields. The power of BogusBiter’s

offensive line may be reduced because some users may just choose to leave a phishing site, but perhaps some security-conscious users are willing to intentionally do some volunteering work to help strike back at the phishers.

(3) *CAPTCHA testing attack*: A phisher may use a CAPTCHA [3] test to evade BogusBiter. CAPTCHA tests are mainly used to prevent automated registrations, but are seldom used in user authentication processes. As shown in our legitimate site experiments, none of the legitimate sites asked a user to do a CAPTCHA test when the set size S is less than 10, and we actually assumed that false positives happened on all those websites. Introducing CAPTCHA testing attacks may decrease the number of phishing victims because the look and feel of the phishing site becomes quite different from that of the targeted legitimate website, and perhaps some users are unable or unwilling to solve CAPTCHAs [54, 158]. Ignoring these disadvantages, a phisher may still want to invoke a CAPTCHA testing attack at either a login page or a login response page.

These attacks may reduce the power of BogusBiter's offensive line, but will not affect the defensive line enabled by BogusBiter. If a phisher invokes the CAPTCHA testing at the login page, the $S - 1$ requests generated by BogusBiter contain the same CAPTCHA answer as that of the original request; therefore, it is difficult for a phisher to tell which credential is entered by a human. If a phisher invokes the CAPTCHA testing at each of the S response pages, we recommend letting BogusBiter to make a replacement so that the CAPTCHA image on the first received response page is used on all the response pages. Therefore, it is still difficult for a phisher to identify the credential entered by a human. If there are legitimate websites that suffer from false positives and meanwhile want to use CAPTCHA testing on each of their response pages after seeing a small set of credential

submissions from BogusBiter, they can simply send back the same CAPTCHA image on each of the S response pages thus will not be affected by this approach.

3.7 Related Work

Basically the various client-side anti-phishing techniques can be classified into three different approaches. The first approach focuses on building tools or toolbars to enhance the security of a login process. Ye and Smith [118] designed a prototype of “Trusted Path” to convey relevant trust signals from a Web browser to a human user. Dhamija and Tygar [19] proposed “Dynamic Security Skins” to allow a legitimate website to prove its identity in a way that is easy for a user to verify but hard for a phisher to spoof. Ross et al. [91] designed PwdHash to transparently produce different passwords for different domains, so that passwords stolen at a phishing site are not useful at a legitimate website. Wu et al. [115] introduced “Web Wallet” to direct an alternative safe path to a user if the user’s intended website does not match the current website. Yee and Sitaker [119] developed Passpet to combine the advantages of several previously devised techniques including petnames, password strengthening, and UI customization. Adida [1] proposed BeamAuth to use a secret token in a URL fragment identifier as a second factor for Web-based authentication. These tools are very helpful, but users must be well trained to use them and must change some of their login habits. Usability is critical to the success of anti-phishing tools [16].

The second approach focuses on improving the accuracy of automatic phishing detection techniques. Chou et al. [17] built SpoofGuard to compute spoof indexes using heuristics and to provide warnings for suspected phishing websites. Recent work by Zhang et al. [132]

and Garera et al. [35] demonstrate that heuristic-based techniques can correctly identify over 90% of phishing pages with about 1% false positives. Fette et al. [26] demonstrated that their machine-learning based techniques can correctly identify over 96% of phishing emails while mis-classifying only 0.1% of legitimate emails. Many other automatic phishing detection tools or toolbars have been developed, and both Firefox 2 and IE 7 have automatic phishing detection as a built-in feature. The evaluation of popular automatic phishing detection tools, toolbars, and Web browser features can be found in [68, 131, 146, 151].

Researchers have also sought to develop nonpreventive anti-phishing approaches. Florêncio and Herley [30] proposed a password rescue scheme that relies on client-side reporting and server-side aggregation to detect and protect stolen credentials. However, this scheme can only statistically make a detection decision after several users become victims, and it also raises privacy concerns by using an extra server to collect user activity information. Parno et al. [84] proposed a Phoolproof anti-phishing mechanism. Although their mechanism eliminates reliance on perfect user behavior, a trusted mobile device must be used to perform mutual authentications. Birk et al. [11] introduced an “active phishing tracing” method, which injects fingerprinted credentials into phishing sites to trace money laundering. Their method can support forensic analyses and enforce judicial prosecutions, but it cannot directly protect phishing victims. Anti-phishing companies such as Cyota (acquired by RSA Security) [179] and Markmonitor [167] have also experimented with injecting special credentials into a phishing site. However, these solutions are less effective than BogusBiter because they neither take the browser integration approach nor enable legitimate websites to detect victims’ stolen credentials.

Finally, there is a related work in “spamming the spammers,” and IBM actually offered

a service to bounce unwanted email back to the computers that sent them [156]. The objective of a spammer is to send junk emails, and IBM's approach intends to offend spammers by consuming their resources. In contrast, the objective of a phisher is to collect real credentials, and our approach intends to make it less likely for phishers to succeed by building both an offensive line and a defensive line.

3.8 Summary

We introduced BogusBiter, a new client-side anti-phishing tool to automatically protect vulnerable users by injecting a relatively large number of bogus credentials into phishing sites. These bogus credentials hide victims' real credentials, and force phishers to verify their collected credentials at legitimate websites. The credential verification actions initiated by phishers, in turn, create opportunities for legitimate websites to detect stolen credentials in a timely manner. BogusBiter is transparent to users and can be seamlessly integrated with current phishing detection and warning mechanisms on Web browsers. We implemented BogusBiter as a Firefox 2 extension and evaluated its effectiveness and usability.

Phishing is a serious security problem today, and phishers are smart, economically motivated, and adaptable. We must therefore actively pursue different approaches and promote the cooperation of different solutions. The effectiveness of BogusBiter depends on many factors, as we discussed in Section 3.6. But we believe its unique approach will make a useful contribution to the anti-phishing research.

Chapter 4

Characterizing Insecure JavaScript Practices on the Web

In this chapter, we present our work on characterizing insecure JavaScript practices. Insecure JavaScript practices may not necessarily result in direct security breaches, but they could definitely cultivate the creation of new attack vectors and greatly increase the risks of browser-based attacks. We present an execution-based measurement approach and the first large-scale measurement study of the insecure JavaScript practices on the Web. Our work sheds light on the insecure JavaScript practices and especially reveals the severity and nature of insecure JavaScript inclusion and dynamic generation practices on the Web.

4.1 Motivation

Security is an important aspect of Web engineering, and it should be taken into serious consideration in the development of high quality Web-based systems [13, 55, 73, 81, 86]. In

many cases, however, security does not receive sufficient attention due to the complexity of Web-based systems, the ad hoc processes of system development, and even the fact that many designers or developers lack security knowledge on Web development techniques. It is not a surprise therefore, that website security breaches are common [22] and Web applications are more susceptible to malicious attacks than traditional computer applications [93].

Browser-based attacks have posed serious threats to the Web in recent years. Exploiting the vulnerabilities in Web browsers [15, 88] or Web applications [41, 53], attackers may directly harm a Web browser's host machine and user through various attacks such as drive-by download [79, 87, 107], cross-site scripting [33, 193], cross-site request forgery [7, 181], and Web privacy attacks [12, 44]. Attackers may even use browsers to indirectly launch large-scale distributed attacks against Web servers [65] or propagate Internet worms [66].

Most of these browser-based attacks are closely tied with JavaScript, which is an interpreted programming language most often used for client-side scripting. JavaScript code embedded or included in HTML pages runs locally in a user's Web browser, and it is mainly used by websites to enhance the interactivity and functionality of their webpages. However, because JavaScript is equipped with a powerful and diverse set of capabilities in Web browsers [28], it has also become the weapon of choice for attackers.

Modern Web browsers impose two restrictions to enforce JavaScript security: the *sandbox* mechanism and the *same-origin* policy. The former limits JavaScript to execute only in a certain environment without risking damage to the rest of the system, while the latter prevents JavaScript in a document of one origin from interacting with another document of a different origin [28, 180]. Unfortunately, most JavaScript-related security vulnerabilities are still the breaches of either of these two restrictions [160]. Some of these vulnerabilities

are due to Web browser flaws, but the majority of them have been attributed to the flaws and insecure practices of websites [181, 184].

A great deal of attention has been paid to the JavaScript-related security vulnerabilities such as cross-site scripting [33, 109, 142, 181, 184] that could directly lead to security breaches. However, little attention has been given to websites' insecure practices of using JavaScript on their webpages. Similar to websites' other insecure practices such as using the customers' social security numbers as their login IDs [24], insecure JavaScript practices may not necessarily result in direct security breaches, but they could definitely cultivate the creation of new attack vectors.

In this work, we present the first measurement study on insecure practices of using JavaScript at different websites. We mainly focus on two types of insecure practices: *insecure JavaScript inclusion* and *insecure JavaScript dynamic generation*. We define the former as the practices of using the `src` attribute of a `<script>` tag to directly or indirectly include a JavaScript file from an external domain into the *top-level document* of a webpage. A top-level document is the document loaded from the URL displayed in a Web browser's address bar. By "directly", we mean that the `<script>` tag belongs to the top-level document, and by "indirectly", we mean that the `<script>` tag belongs to a sub-level frame or iframe document whose origin is the same as that of the top-level document. We define the latter as the practices of using dangerous techniques such as the `eval()` function to dynamically generate new scripts. Both types of insecure practices create new vectors for attackers to inject malicious JavaScript code into webpages and launch attacks such as cross-site scripting and cross-site request forgery.

The primary objective of our work is to examine the severity and nature of these two

types of insecure JavaScript practices on the Web. To achieve this goal, we devised an execution-based measurement approach. More specifically, we instrumented the Mozilla Firefox 2 Web browser and visited the homepages of 6,805 popular websites in 15 different categories. The instrumented Firefox non-intrusively monitors the JavaScript inclusion and dynamic generation activities on those webpages, and it precisely records important information for offline analysis.

Our measurement results reveal that insecure JavaScript inclusion and dynamic generation practices are widely prevalent among websites. At least 66.4% of the measured websites have the insecure practices of including scripts from external domains into the top-level documents of their homepages. Over 74.9% of the measured websites use one or more types of JavaScript dynamic generation techniques, and insecure practices are quite common. For example, `eval()` function calls exist at 44.4% of the measured websites. Using the `document.write()` method and the `innerHTML` property is much more popular than using the relatively secure method of creating JavaScript elements via DOM (Document Object Model) [191] methods. Our results also show that around 94.9% of the measured websites register various event handlers on their homepages, implying that the captured insecure JavaScript practices in inclusion and dynamic generation are likely conservative estimates.

The main contribution of our work is threefold. First, we introduce a browser instrumentation framework that enables us to capture essential JavaScript execution behavior on webpages. Not only can this framework measure the insecure JavaScript practices, it can also examine other JavaScript execution characteristics such as function call patterns and code (de)obfuscation activities. Second, we present a classification method to analyze

and classify different types of dynamically generated JavaScript code. By extracting the AST (abstract syntax tree) trees of scripts and performing AST signature creation and matching, our classification method can effectively assist us in understanding the structural information of the hundreds of thousands of dynamically generated scripts. Third, our measurement study sheds light on the insecure JavaScript practices and especially reveals the severity of insecure JavaScript inclusion and dynamic generation practices on the Web. Our in-depth analysis further indicates that safe alternatives to these insecure practices do exist in common cases. We therefore suggest website developers and administrators pay serious attention to these insecure engineering practices and use safe alternatives to avoid them.

The remainder of this chapter is structured as follows. Section 4.2 explains why the two types of JavaScript practices are insecure. Section 4.3 introduces our measurement and analysis methodologies. Section 4.4 describes the data set of this study. Section 4.5 presents and analyzes our measurement results. Section 4.6 reviews related work, and finally, Section 4.7 summarizes this work.

4.2 Background

In the same-origin policy, the origin of a document is defined using the protocol, domain name, and port of the URL from which the document is loaded. It is important to realize that this policy does not limit the origin of a script itself. Although JavaScript code cannot access another document loaded from a different origin, it can fully access the document in which it is embedded or included even when the code has a different origin than the

document [28]. Including scripts from an external domain into the top-level document of a webpage is very dangerous because it grants the scripts the maximum permissions allowed to control the webpage and the browser window. Therefore, if the author of a script file or the administrator of a script hosting site is insincere or irresponsible, insecure JavaScript inclusion practices could lead to serious security and privacy breaches. Moreover, script hosting sites could become attractive targets of attacks, especially when their JavaScript files are included by multiple websites. To lower the potential risks, websites should avoid external JavaScript inclusion by using internal JavaScript files from the same sites when possible. Otherwise if external inclusion is really inevitable, for example some advertising sites or traffic analysis sites may necessitate it [82], external included scripts should be retrieved using HTTPS connections and should be restricted within a sub-level HTML frame or iframe document whose origin is different from that of the top-level document.

The `eval()` function takes a string parameter and evaluates it as JavaScript code. This function is dangerous because it executes the passed script code with the privileges of the function's caller [145]. Therefore, attackers may endeavor to inject malicious code into the evaluated string in order to take advantage of this capability. Meanwhile, since scripts are dynamically generated and evaluated, it is very challenging to effectively filter out maliciously injected code [51, 88, 120]. `Eval()` should be avoided ¹ if at all possible, and its safe alternatives should be used [133, 145]. Other JavaScript dynamic generation techniques such as using the `document.write()` function and the `innerHTML` property also pose similar security risks, as discussed in Section 4.5.

Once attackers have successfully exploited these insecure practices and injected their

¹Searching “eval is evil” on the Web for many discussions.

malicious JavaScript code, they can easily launch severe attacks such as cross-site scripting and cross-site request forgery. These attacks can be used to conduct many malicious activities such as account hijacking, user behavior tracking, denial of service attacking, and website defacing. Therefore, insecure engineering practices of using JavaScript should be thoroughly investigated, their risks should be highlighted to Web developers, and safe alternatives should be used to avoid them.

4.3 Methodology

We devised an execution-based measurement approach to study the insecure JavaScript practices on the Web. Our strategy is to first use an instrumented Web browser to obtain actual JavaScript execution trace information on different webpages, and then use offline analysis to characterize and understand various JavaScript practices. An alternative approach is to simply perform static analysis on webpages. However, this approach suffers from the problem of undecidability and is unable to precisely determine which scripts will be generated and executed. In contrast, our approach allows us to effectively capture the dynamics of webpages and JavaScript code in their real runtime environments. Figure 4.1 gives an overview of our instrumentation framework and analysis toolkit.

4.3.1 Instrumentation Framework

To achieve an accurate and efficient measurement, we employed the source code instrumentation technique and instrumented the most popular open source Web browser—Mozilla Firefox. Our instrumentation method is similar to program tracing, which is a well-known approach for monitoring program behavior and measuring program performance. We fol-

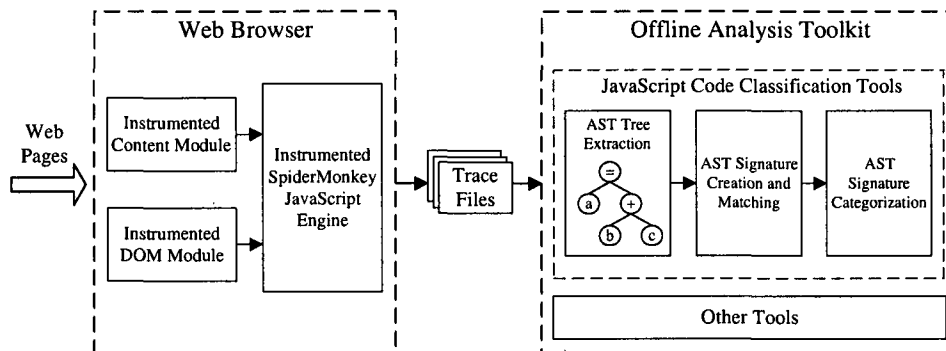


Figure 4.1: Overview of the instrumentation framework and analysis toolkit.

lowed a few rules suggested in [6] to minimize instrumentation overhead. More specifically, we attempted to insert less instrumentation code and place the code only at necessary points with low execution frequency.

We mainly instrumented three modules of Firefox 2 source code: the JavaScript engine, the content module, and the DOM module. Firefox uses SpiderMonkey as its JavaScript engine [183]. SpiderMonkey JavaScript engine is written in C programming language and is a relatively independent module in Firefox. The major interface between SpiderMonkey and other modules in Firefox is the SpiderMonkey JSAPI [161]. JSAPI facilitates other modules in Firefox to use the core JavaScript data types and functions of SpiderMonkey, and it also allows other modules to expose some of their objects and functions to JavaScript code.

Inside the SpiderMonkey, our instrumented code written in C consists of three parts. First, eight trace logging functions were integrated into the JSAPI interface. These functions facilitate the trace collection in a consistent manner, recording various information such as script text, function calls, and event handler registrations. Second, we added code to the byte-code interpreter of SpiderMonkey so that we can record the execution informa-

tion of any global scripts and function scripts. Third, we instrumented the object system implementation of SpiderMonkey to monitor the calls to the `eval()` function and collect both the calling context information and the evaluated content information.

The trace files generated in the above instrumentation points enable us to analyze the practices of JavaScript inclusion and the practices of JavaScript dynamic generation using `eval()`. We also needed to monitor the practices of other JavaScript dynamic generation techniques. Originally we attempted to fulfill this task by still instrumenting inside the SpiderMonkey and monitoring the engine's native callbacks to the content and DOM modules. However, we found that this approach induces high overhead and could only record partial information. Therefore, we decided to directly instrument the content module and the DOM module of Firefox.

In the content module of Firefox, we integrated C++ code to measure the other three types of JavaScript dynamic generation techniques. We instrumented the `document.write()` method² and the method for setting the `innerHTML` property of an HTML element to track their invocations. Both techniques can be used to add new content to an HTML document, and the added content may contain new JavaScript code. We also added code to monitor the method for replacing, inserting or appending a new DOM element, which could be created by using DOM methods such as `document.createElement()` and `document.createTextNode()`. Our instrumentation code can identify the script type of elements and record their source and text information. Other techniques such as the `insertAdjacentHTML()` method or the `outerHTML` property are supported in the Internet Explorer Web browser only, and we cannot measure them in Firefox.

²In this work, it also includes the `document.writeln()` method.

In the DOM module and the content module, we added C++ code to measure various event handler registration techniques supported in Firefox. Event handlers can be triggered by user interaction or timer events. We collected event handler registration information to show that further JavaScript inclusion and execution could happen and our captured insecure practices are likely conservative estimates. Event handler registration and other aspects of information described above are written into a set of six different trace files to assist our offline analysis.

Since many internal user interface components of Firefox also heavily use JavaScript, special care is needed to ensure that the above instrumentation code only records the JavaScript execution activities of a visited webpage. Our code checks the JSPrincipals [163] information of an object or script to guarantee this requirement. We also ensured that our instrumentation code only monitors and records essential information and does not change the execution logic of Firefox and SpiderMonkey.

4.3.2 Analysis Toolkit

We took an offline analysis approach so that we can sufficiently analyze the trace information without interfering with the actual measurement process. We developed an offline analysis toolkit that consists of a set of tools written in approximately 5,000 lines of Java code, 200 lines of C code, 500 lines of Linux shell script code, and 300 lines of Matlab script code. About half of the tools are used for classifying dynamically generated JavaScript code, and the others are used for processing trace records and calculating statistical information. The detailed description of the JavaScript code classification tools is as follows.

The motivation for developing these classification tools is to automate the challenging

task of understanding a large number of dynamically generated JavaScript code. To achieve this goal, we explored the concepts in software engineering and developed an AST (abstract syntax tree)-based classification method. As illustrated in Figure 4.1, the key idea is to first extract the AST trees of scripts, then create and match AST signatures, and finally merge signatures into different categories. We devised such an AST-based approach in that ASTs have been demonstrated effective in program understanding [8, 110].

The AST tree extraction tool is a standalone C program that embeds the SpiderMonkey 1.7 [183]. This is the same version of the SpiderMonkey as used in our instrumented Firefox 2 Web browser. Therefore, our extraction tool can create a token stream and parse the stream into a syntax tree for a script in the same manner as in the instrumented Firefox. The tool finally constructs the essential structure of a syntax tree as an AST tree and writes the tree into an XML file to facilitate further comparison.

We applied *top-down tree matching* techniques to perform AST signature creation and matching, and the high-level procedure is illustrated in Figure 4.2. First, an empty AST signature set S is initialized. Next, for each AST tree in the XML files, its top N level structure is used to generate an AST signature, denoted as *thisSig*. Then, top-down tree comparisons are made to seek a match between the *thisSig* and an existing signature in the set S . If a match exists, this procedure keeps a record of the related information, otherwise, the *thisSig* is added to the set S as a new AST signature. Finally, this procedure returns the signature set S as its output.

To be accurate and representative, an AST signature keeps the name and type information of an operator node, but it only keeps the type information of an operand. Top-down tree matching techniques can capture the key structural differences between trees, and they

```

SigCreateMatch (XMLfiles,  $N$ )
1. Initialize an empty AST signature set  $S$ ;
2. for each AST tree in the XML files do
3.   thisSig=the top  $N$  level structure of the AST tree;
4.   if thisSig matches an existing signature in  $S$  then
5.     Record the information of this matching;
6.   else
7.      $S = S \cup \{\text{thisSig}\}$ ;
8.   endif
9. endfor
10. return the result set  $S$ ;

```

Figure 4.2: High-level AST signature creation and matching procedure.

have been used in several Web-related projects [89, 126, 128]. The comparison algorithm used in line 4 of this procedure is adapted from the STM (simple tree matching) algorithm presented in [117]. STM is an efficient top-down tree distance comparison algorithm, and our adaptation is to only compare the top N levels of trees. As shown in Section 4.5, such an adaptation is effective in striking a good balance between retaining the accuracy and reducing the total number of signatures.

The AST signature categorization tool was developed to further merge AST signatures into different categories. We defined categories according to different types of JavaScript expressions and statements such as arithmetic expressions and assignment statements. Such a categorization can help us to understand the use purposes of JavaScript code from a programming language perspective. This tool is especially useful for analyzing dynamically generated scripts, most of which have specific use purposes in terms of programming language functionality as revealed in our analysis.

Category	com	org	gov	net	edu	cc	other	Total
arts	417	16	0	27	1	39	0	500
business	430	7	10	4	0	49	0	500
computers	432	29	1	21	1	15	1	500
games	428	13	0	43	0	14	2	500
health	277	107	41	8	33	30	4	500
home	415	28	22	14	2	18	1	500
news	412	24	6	12	3	43	0	500
recreation	409	19	12	19	0	40	1	500
reference	116	17	11	4	192	158	2	500
regional	292	23	21	6	3	152	3	500
science	209	96	68	8	47	64	8	500
shopping	479	2	0	2	0	17	0	500
society	302	84	34	11	3	58	8	500
sports	403	13	0	21	0	62	1	500
world	199	15	1	23	0	262	0	500
Total	5220	493	227	223	285	1021	31	7500
Uniq-Total	4727	445	170	212	276	950	25	6805

Table 4.1: Category breakdown by top-level domain.

4.4 Data Set

To obtain a representative data set, we followed a similar method as used in [60] and selected top websites listed by Alexa.com [136]. We chose 15 categories and then top 500 sites from each of these categories. Table 4.1 gives the breakdown of 15 categories by DNS top-level domain (TLD). Since some sites appear in multiple categories, the total number of unique sites is 6,805 in our study. This number is over five times larger than that in [60], and we also only visited the homepages of those sites so that we can have a consistent measurement. Meanwhile, measuring the insecure JavaScript practices on homepages is sufficient to illustrate the severity of the problem. Table 4.1 shows that the majority of the 6,805 sites come from the .com TLD and the country code (denoted as the cc) TLD. The former contributes 4,727 unique sites and the latter contributes 950 unique sites.

The execution of JavaScript on a webpage can be roughly divided into two phases: the document loading and parsing phase and the event-driven phase [28]. When the document

loading and parsing phase ends, the event-driven phase starts and event handlers can be asynchronously executed in response to various user interaction and timer events. In our study, we developed a browser extension to automatically visit each of the 6,805 webpages using our instrumented Firefox Web browser. On each page, our browser extension waits for the end of the document loading and parsing phase and then stays in the event-driven phase for 10 seconds. Our browser extension has no intention to trigger the execution of any specific event handlers on a page. This is because the event handlers registered on different webpages are very diverse, and it is difficult to trigger their executions in a consistent manner. Therefore, the JavaScript execution data set collected in our measurement study covers the whole document loading and parsing phase and 10 seconds of the event-driven phase for each of the 6,805 homepages. The data set was collected in the second week of July 2008.

4.5 Results and Analysis

We present and analyze our measurement results in this section. We first briefly present the results on JavaScript presence. Then, we detail the results on the insecure practices of JavaScript inclusion and dynamic generation. Finally, we give a short summary of the results on event handler registrations.

4.5.1 Overall JavaScript Presence

Table 4.2 lists the results of overall JavaScript presence for the 6,805 measured homepages. We use *JS* to represent any JavaScript code, and we use *DJS* to represent the JavaScript code that is dynamically generated by using one of the four dynamic generation techniques

Category/ TLD	Pages with any JS			Pages with DJS
	embedded JS	included JS	Total	
arts	484(96.8%)	483(96.6%)	491(98.2%)	437(87.4%)
business	482(96.4%)	473(94.6%)	492(98.4%)	380(76.0%)
computers	471(94.2%)	465(93.0%)	484(96.8%)	374(74.8%)
games	471(94.2%)	473(94.6%)	488(97.6%)	375(75.0%)
health	467(93.4%)	451(90.2%)	481(96.2%)	330(66.0%)
home	479(95.8%)	471(94.2%)	487(97.4%)	389(77.8%)
news	477(95.4%)	475(95.0%)	483(96.6%)	430(86.0%)
recreation	477(95.4%)	467(93.4%)	487(97.4%)	389(77.8%)
reference	455(91.0%)	443(88.6%)	476(95.2%)	286(57.2%)
regional	479(95.8%)	457(91.4%)	492(98.4%)	401(80.2%)
science	421(84.2%)	405(81.0%)	449(89.8%)	274(54.8%)
shopping	487(97.4%)	486(97.2%)	493(98.6%)	393(78.6%)
society	441(88.2%)	435(87.0%)	466(93.2%)	329(65.8%)
sports	492(98.4%)	482(96.4%)	496(99.2%)	456(91.2%)
world	481(96.2%)	438(87.6%)	489(97.8%)	377(75.4%)
com	4551(96.3%)	4504(95.3%)	4629(97.9%)	3838(81.2%)
org	401(90.1%)	378(84.9%)	422(94.8%)	247(55.5%)
gov	150(88.2%)	137(80.6%)	160(94.1%)	75(44.1%)
net	194(91.5%)	189(89.2%)	204(96.2%)	153(72.2%)
edu	239(86.6%)	223(80.8%)	250(90.6%)	122(44.2%)
cc	863(90.8%)	817(86.0%)	902(94.9%)	654(68.8%)
other	23(92.0%)	22(88.0%)	24(96.0%)	9(36.0%)
All	6421(94.4%)	6270(92.1%)	6591(96.9%)	5098(74.9%)

Table 4.2: JavaScript presence by category and top-level domain.

measured in our instrumented Firefox Web browser. The *embedded JS* indicates that the executed JavaScript code is embedded within an HTML document, and the *included JS* indicates that the executed JavaScript code is included from a separate file.

Overall, JavaScript execution has been widely observed on 6,591(96.9%) homepages. Both the JS embedding and JS inclusion are very common, and they are practiced on 6,421 and 6,270 pages, respectively. The percentage of webpages containing JavaScript execution within a category ranges from 89.8% for science to 99.2% for sports, and the percentage of webpages containing JavaScript execution within a TLD ranges from 90.6% for .edu to 97.9% for .com. JavaScript dynamic generation is also very popular, and there are 5,098 (74.9%) sites containing DJS on their homepages. For the DJS presence within a category,

the lowest percentage is 54.8% for science, and the highest percentage is 91.2% for sports. For the DJS presence within a TLD, the highest percentage is 81.2% for .com, and the lowest percentage is 36.0% for other domains such as .mil and .info.

4.5.2 Insecure JavaScript Inclusion

Among all the 6,270 webpages with the included JS, we identify and analyze insecure practices of JavaScript inclusion. Note that we defined the insecure JavaScript inclusion as the practices of using the `src` attribute of a `<script>` tag to directly or indirectly include a JavaScript file from an external domain into the top-level document of a webpage. Keeping JavaScript code separate from HTML markups is actually a good engineering practice, advocated especially in the unobtrusive JavaScript programming paradigm [28, 189]. Therefore, there is no need to analyze the good practices of including JavaScript files from the same host or domain, and we only focus on the insecure inclusion practices.

4.5.2.1 Results and Analysis

To our surprise, insecure JavaScript inclusion is very prevalent. Around 66.4% (4,517 out of 6,805) of websites directly or indirectly include JavaScript files from external domains into the top-level documents of their homepages. Note that our analysis tool applies a conservative standard to compare the domain name of a JavaScript file and that of its including homepage. Two domain names are regarded as different only if, after discarding their top-level domain names (e.g., .com) and the leading name “www” (if existing), they do not have any common sub-domain name³. Therefore, this 66.4% result is basically an

³For example, two domain names `www.d1sub2.d1sub1.d1tld` and `d2sub3.d2sub2.d2sub1.d2tld` are regarded as different only if the intersection of the two sets `{d1sub2, d1sub1}` and `{d2sub3, d2sub2, d2sub1}`

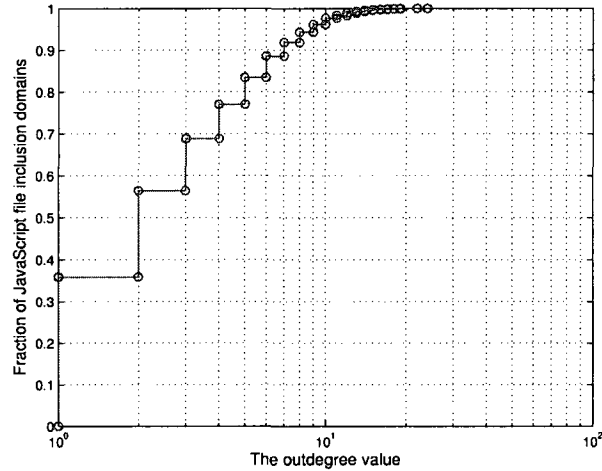


Figure 4.3: Cumulative distribution of the 4,517 JavaScript file inclusion domains in terms of their outdegree values.

objective estimate of the severity of insecure JavaScript inclusion practices.

After further analyzing the domain name relationship between JavaScript file inclusion sites and JavaScript file hosting sites, we found that those 4,517 sites include JavaScript files from a diverse set of 1,985 external domains. We can use a directed graph to characterize the domain name relationship between these sites. Different vertices represent different domain names, and a direct edge from vertex A to vertex B means that the homepage in domain A includes at least one JavaScript file from domain B. Therefore, 4,517 vertices have a greater than zero outdegree value, and 1,985 vertices have a greater than zero indegree value.

Figure 4.3 illustrates the CDF (cumulative distribution function) of the 4,517 JavaScript file inclusion domains in terms of their outdegree values. We can see that approximately 43.6% of the 4,517 sites include JavaScript files from at least three external domains. While the mean value of outdegree is 3.1, the maximum value of outdegree reaches 24. These

is empty.

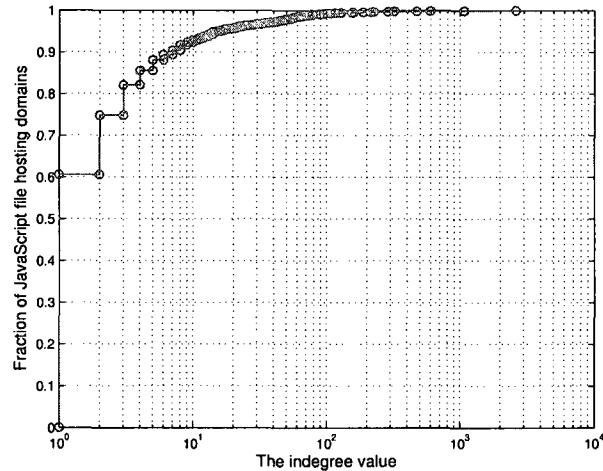


Figure 4.4: Cumulative distribution of the 1,985 JavaScript file hosting domains in terms of their indegree values.

results indicate that not only 66.4% of measured sites are at the risk of having their homepages under the control of the included JavaScript code, but many of them also face higher risks from multiple sources.

From a different perspective, Figure 4.4 depicts the CDF of the 1,985 JavaScript file hosting domains in terms of their indegree values. We can observe two interesting phenomena. On the one hand, JavaScript files in approximately 60.6% of the hosting domains are only included by one of our visited homepages. On the other hand, JavaScript files in approximately 7.7% of the hosting domains are included by at least 10 of our visited homepages, and JavaScript files in 14 sites are even included by at least 100 of our visited homepages. The mean value of indegree is 7.2, but the maximum value of indegree reaches a very high value of 2,606. After inspecting those 14 high-profile JavaScript file hosting domains and many other low-profile domains, we found that a few of them are popular traffic analysis service sites and advertising servers. However, most of them are the kind of “hidden” sites that provide nothing on their root URLs but just point to some stored

JavaScript files using URL paths. Understanding the properties of those sites is beyond the scope of this work, but what we need to emphasize is that external JavaScript file hosting sites, especially those high-profile ones, create new vectors for large-scale browser-based attacks. Even a single compromised JavaScript file could directly cause security breaches on thousands of websites.

Among the 4,517 sites that include JavaScript files from external domains, we also observed that 125 sites only use the HTTPS protocol to retrieve JavaScript files and 138 sites use both the HTTP protocol and the HTTPS protocol to retrieve different JavaScript files. In total, there are 263 sites using HTTPS to include scripts from 72 JavaScript file hosting sites. These observations imply that some JavaScript file hosting sites do provide the secure transmission service for accessing their hosted JavaScript files, and some of our measured sites do use this service. However, this secure JavaScript transmission service is not popular. Only 3.6% (72 out of 1,985) of the JavaScript file hosting sites provide the service, and only 5.8% (263 out of 4,517) of the JavaScript file inclusion sites use the service. Also note that HTTPS protects data in transit, but it does not guarantee that a JavaScript file is uncompromised in a hosting site.

In contrast to these 4,517 sites, we did find that there are 324 other sites, in which an external included JavaScript file is always restricted within a sub-level HTML frame or iframe document whose origin is different from that of the top-level document. This observation implies that some sites do limit the control of external included JavaScript code within sub-level documents and provide a protection to the top-level documents of their homepages. However, such a relatively secure practice is exclusively followed by only 324 measured sites, and those 4,517 sites still use a very insecure way to include external

JavaScript files.

4.5.2.2 Safe Alternatives to Insecure Inclusion

Our results show that insecure JavaScript inclusion is widely practiced by the majority (66.4%) of our measured sites. Our in-depth analysis on the domain name relationship between JavaScript file inclusion sites and hosting sites further reveals the severity and nature of those insecure practices. Although HTTPS and sub-level documents are used by a small portion of sites to enhance the security of external JavaScript file inclusion, we believe that the majority of measured JavaScript file inclusion sites and hosting sites have not paid sufficient attention to the potential risks of insecure JavaScript inclusion. For JavaScript file inclusion sites, we suggest them (1) avoid external JavaScript inclusion by using internal JavaScript files from the same sites, if at all possible; (2) restrict the permission of external included scripts by placing them within a sub-level HTML frame or iframe document whose origin is different from that of the top-level document, if external inclusion is really inevitable; and (3) retrieve external JavaScript files using HTTPS connections, if the HTTPS service is available. The third suggestion needs a hosting site to provide the HTTPS service for accessing its JavaScript files, but the first two suggestions can be easily adopted by JavaScript file inclusion sites.

4.5.3 Insecure JavaScript Dynamic Generation

Since 74.9% of measured sites (5,098 out of 6,805) contain DJS scripts on their homepages, we now characterize all the DJS scripts based on their generation techniques and analyze insecure practices.

Category/ TLD	eval- generated	write- generated	innerHTML- generated	DOM- generated
arts	258(51.6%)	403(80.6%)	76(15.2%)	83(16.6%)
business	253(50.6%)	295(59.0%)	73(14.6%)	56(11.2%)
computers	205(41.0%)	307(61.4%)	55(11.0%)	55(11.0%)
games	203(40.6%)	327(65.4%)	58(11.6%)	57(11.4%)
health	190(38.0%)	276(55.2%)	35(7.0%)	34(6.8%)
home	240(48.0%)	357(71.4%)	57(11.4%)	73(14.6%)
news	314(62.8%)	412(82.4%)	161(32.2%)	110(22.0%)
recreation	229(45.8%)	310(62.0%)	67(13.4%)	57(11.4%)
reference	144(28.8%)	214(42.8%)	44(8.8%)	22(4.4%)
regional	258(51.6%)	337(67.4%)	97(19.4%)	58(11.6%)
science	137(27.4%)	234(46.8%)	39(7.8%)	35(7.0%)
shopping	245(49.0%)	307(61.4%)	37(7.4%)	38(7.6%)
society	163(32.6%)	283(56.6%)	42(8.4%)	42(8.4%)
sports	322(64.4%)	424(84.8%)	114(22.8%)	95(19.0%)
world	212(42.4%)	341(68.2%)	92(18.4%)	55(11.0%)
com	2359(49.9%)	3359(71.1%)	724(15.3%)	656(13.9%)
org	109(24.5%)	195(43.8%)	25(5.6%)	22(4.9%)
gov	32(18.8%)	50(29.4%)	9(5.3%)	9(5.3%)
net	77(36.3%)	135(63.7%)	26(12.3%)	21(9.9%)
edu	50(18.1%)	92(33.3%)	17(6.2%)	7(2.5%)
cc	393(41.4%)	558(58.7%)	130(13.7%)	79(8.3%)
other	4(16.0%)	7(28.0%)	3(12.0%)	0(0.0%)
All	3024(44.4%)	4396(64.6%)	934(13.7%)	794(11.7%)

Table 4.3: DJS presence by category and top-level domain.

4.5.3.1 DJS Presence by Category and TLD

Table 4.3 lists the overall DJS presence by category and TLD for the four different DJS generation techniques. We can see that the `eval()` function and the `document.write()` method are widely used on 44.4% and 64.6% of webpages, respectively. In contrast, the `innerHTML` property and the DOM methods (i.e., replacing, inserting or appending a new created script element) are only used on 13.7% and 11.7% of webpages, respectively. It is also interesting to notice that the categories with the highest DJS presence values are news and sports for all the four generation techniques. The TLDs with the highest DJS presence values are .com, .net, and country code domains. These results indicate that JavaScript dynamic generation is more likely to be used on those sites that have more dynamic contents.

4.5.3.2 DJS Instance Summary

We now examine the generated DJS instances on each webpage. A DJS instance is identified in different ways for different generation techniques. For the `eval()` function, the whole evaluated string content is regarded as a DJS instance. Within the written content of the `document.write()` method and the value of the `innerHTML` property, a DJS instance can be identified from three sources: (1) between a pair of `<script>` and `</script>` tags; (2) in an event handler specified as the value of an HTML attribute such as `onclick` or `onmouseover`; and (3) in a URL that uses the special `javascript:protocol` specifier [28]. For the DOM methods, each new script element is identified as a DJS instance.

Table 4.4 gives a summary of DJS instances for both the document loading and parsing phase, denoted as the pre-onload phase, and the event-driven phase, denoted as the post-onload phase. The two numbers in each table cell represent the data for the pre-onload and post-onload phases, respectively. The data in the second row of the table gives the total number of DJS instances identified in the two execution phases for the four different techniques. The data in the third row of the table gives the total number of webpages on which those DJS instances are identified. The IPP in the last three rows of the table stands for the “Instance Per Page”.

It is evident that the `eval()` function generates the largest number of DJS instances in both phases (194,676 in the pre-onload phase and 22,632 in the post-onload phase). The mean value of IPP for `eval`-generated DJS instances is 65.2 in the pre-onload phase and 62.3 in the post-onload phase. The maximum value of IPP for `eval`-generated DJS instances reaches 2,543 in the pre-onload phase and 6,350 in the post-onload phase. These

Summary	eval-generated	write-generated	innerHTML-generated	DOM-generated
total number of DJS instances	194676/ 22632	67446/ 519	28717/ 6626	1370/ 557
total number of pages	2986/ 363	4385/ 63	844/ 187	680/ 260
mean value of IPP	65.2/ 62.3	15.4/ 8.2	34.0/ 35.4	2.0/ 2.1
maximum value of IPP	2543/ 6350	1053/ 160	5001/ 1403	13/ 25
standard deviation of IPP	174.4/ 367.2	41.6/ 20.9	184.3/ 134.1	1.7/ 2.7

Table 4.4: DJS instance summary for pre-onload/post-onload phases.

numbers indicate that `eval()` may be misused or abused. The `document.write()` method also generates a large number of DJS instances in the pre-onload phase, but it only generates 519 DJS instances on 63 pages in the post-onload phase. Calling `document.write()` in post-onload phase is usually not desirable because it will overwrite the current document with the written content. In both phases, the `innerHTML` property also generates a large number of DJS instances, while DOM methods generate much fewer DJS instances.

For the four JavaScript dynamic generation techniques, Figures 4.5(a) to 4.5(d) further illustrate the cumulative distribution of the webpages in terms of IPP. In each of these four figures, the “o” curve is for the pre-onload phase and the “*” curve is for the post-onload phase. Note that the total number of pages is different for the two phases (as shown in the third row of Table 4.4), and we present the two curves together for ease of comparison. We can see that the indication of misuse or abuse is especially evident for the `eval()` function. While the majority (about 60%) of webpages have 10 or less eval-generated DJS instances, nearly 17% and 11% of webpages have 100 or more eval-generated DJS instances for the pre-onload phase and the post-onload phase, respectively.

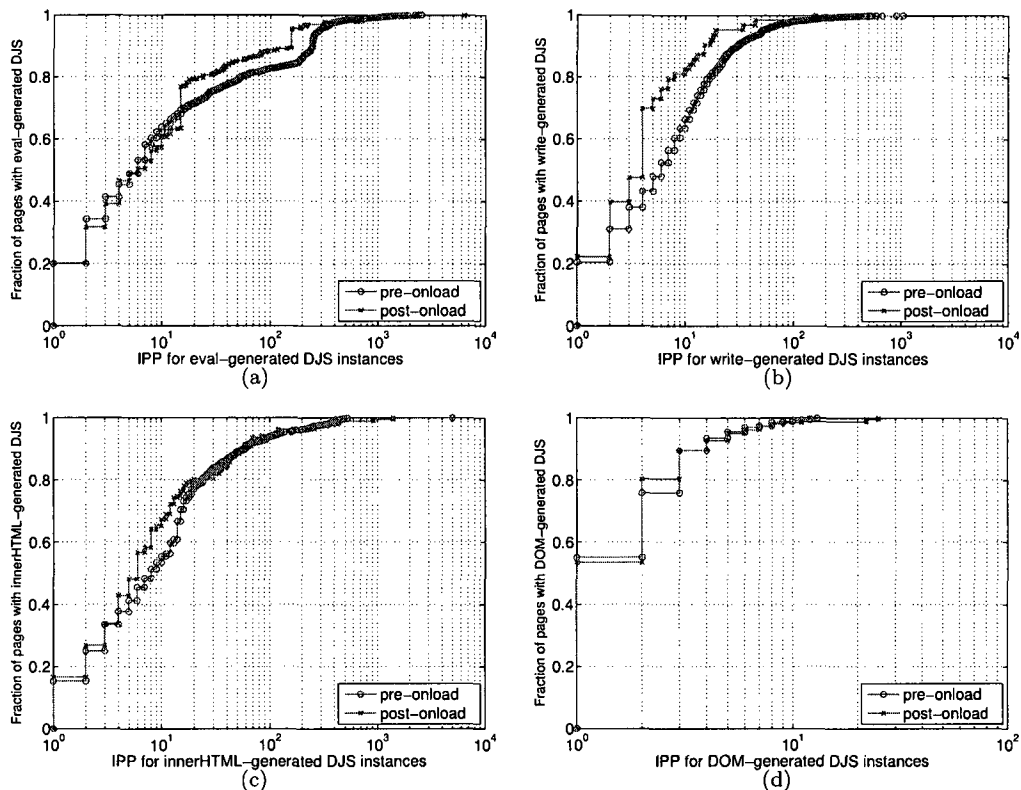


Figure 4.5: Cumulative distribution of the webpages in terms of IPP (Instance Per Page) for (a) eval-generated, (b) write-generated, (c) innerHTML-generated, and (d) DOM-generated DJS instances.

4.5.3.3 Structural Analysis of eval-generated DJS

The prevalence of DJS on various categories of webpages and the high IPP values motivate us to further understand the use purposes of the large number of DJS instances. Using our JavaScript code classification tools, we now uncover the use purposes of eval-generated DJS instances in terms of programming language functionality.

From the total 217,308 (both the pre-onload phase and the post-onload phase) eval-generated DJS instances, 217,308 AST trees are extracted by our AST tree extraction tool. The maximum height of these AST trees is 19. Figure 4.6 shows the cumulative distribution

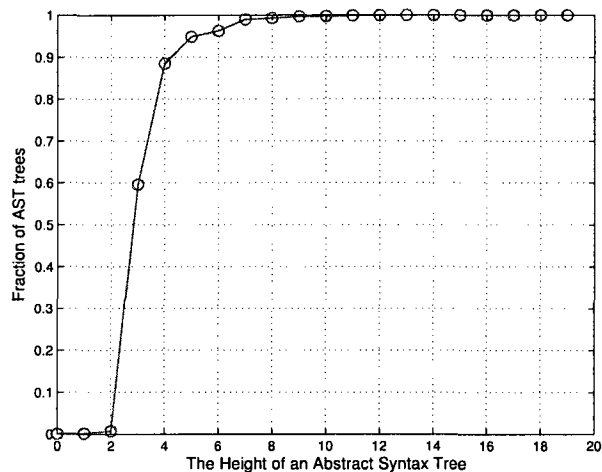


Figure 4.6: Cumulative distribution of the AST trees in terms of the height of an AST tree.

of the AST trees in terms of the height of an AST tree. Nearly 90% of AST trees have a height less than or equal to 4. Therefore, we selected $N = 4$ as the input parameter (Figure 4.2) and used the top-four level structure of AST trees to create and match AST signatures. A total number of 647 AST signatures are created and matched from the 217,308 AST trees. These 647 AST signatures capture the essential structural information of the 217,308 AST trees, and they greatly facilitate our further analysis. Using more levels of AST tree structure is unnecessary because lower-level AST tree nodes only contain less important structural information.

Finally, AST signatures with the same programming language functionality are merged into the same category by using our AST signature categorization tool. For example, two AST signatures representing two types of function calls with different number or type of parameters are merged into the same *function calls* category. Table 4.5 lists the final 17 categories of DJS instances classified from the 647 AST signatures, and in turn from the 217,308 DJS instances.

Category	Presence in Pages	Number of DJS instances	Average DJS length
parse error	20(0.7%)	111(0.05%)	1175.5
empty content	124(4.1%)	291(0.13%)	0.0
simple expression	1209(40.0%)	134251(61.8%)	13.9
arithmetic expression	12(0.4%)	158(0.1%)	67.9
relational expression	79(2.6%)	3246(1.5%)	31.7
logical expression	159(5.3%)	3249(1.5%)	75.8
object/array literal	265(8.8%)	4798(2.2%)	623.0
other expression	126(4.2%)	5789(2.7%)	18.5
variable declarations	98(3.2%)	411(0.2%)	586.0
function declarations	1157(38.3%)	1929(0.9%)	13380.7
assignment statements	1289(42.6%)	42015(19.3%)	51.9
function calls	527(17.4%)	2733(1.3%)	368.9
method calls	561(18.6%)	2062(0.9%)	75.9
object/array creations	29(1.0%)	212(0.1%)	41.3
conditional statements	181(6.0%)	9371(4.3%)	519.1
try-catch statements	1075(35.5%)	4127(1.9%)	51.7
mixed statements	910(30.1%)	2555(1.2%)	6884.1

Table 4.5: The 17 categories of eval-generated DJS instances.

We can see that 0.05% of the DJS instances have *parse error* when AST trees are extracted, and 0.13% of the DJS instances have *empty content*. The majority (around 98.6%) of the eval-generated DJS instances are classified into the 14 categories from *simple expression* to *try-catch statements*. The DJS instances in these 14 categories all have specific use purposes in terms of programming language functionality. Only 1.2% of the DJS instances have mixed programming language functionalities, and they are classified into the last category of *mixed statements*. The generated DJS instances in the last 15 categories are either various expressions (from *simple expression* to *other expression*) or various statements (from *variable declarations* to *mixed statements*). In general, a JavaScript expression is used only to produce a value, while a JavaScript statement normally has side effects and is often used to accomplish some tasks.

4.5.3.4 Safe Alternatives to eval()

To further understand whether using eval() is necessary in these different categories, we randomly sampled and inspected both the content and the calling context of 700 DJS instances. We sampled 200 DJS instances from the simple expression category and 200 DJS instances from the assignment statements category. These two categories have the largest numbers of DJS instances, accounting for 61.8% and 19.3%, respectively, of all the eval-generated DJS instances. The remaining 300 DJS instances are sampled from the other 15 categories, with each of them contributing 20 instances.

In at least 70% of the sampled cases, the eval() function is misused or abused while safe alternatives can be easily identified. Here we illustrate three representative sampled cases. The first one is: `this.homePos = eval("0" + this.dirType + this.dim)`, in which a string simple expression "0-500" is generated. Indeed, such a kind of string concatenation directly generates a string value, and using eval() is redundant. The second one is: `var ff_nav=eval("nav_"+tt[i][1])`, in which a variable name "nav.20912" is dynamically accessed. A safe alternative is using the JavaScript *window* object to directly access the variable: `var ff_nav=window["nav_"+tt[i][1]]`. The third one is: `var responses = eval(o.responseText)`, in which the response content of an XMLHttpRequest [192] is directly evaluated. This practice is used in many of our sampled cases to convert a *responseText* into a JSON object. However, since malicious JavaScript code could be injected into the *responseText*, it would be better to use a JSON parser rather than the eval() function to perform such a transformation [162]. The other 30% of the sampled cases usually have complex calling context, so we do not further identify their safe alternatives.

Technique and Type		Presence in Pages	Number of DJS instances	Avg. length	Main usage
write	jscode	4000	26125	77	JS inclusion
	eventhandler	1773	38650	39	function call
	jsprotocol	501	3190	45	function call
innerHTML	jscode	120	503	262	JS inclusion
	eventhandler	747	31267	60	function call
	jsprotocol	336	3573	33	function call
DOM	src	779	1866	-	JS inclusion
	text	33	61	623	assignment

Table 4.6: Structural analysis of DJS instances generated by the `document.write()` method, `innerHTML` property, and DOM methods.

We suggest that `eval()` should be avoided if at all possible. In addition to the safe alternatives exemplified above, DOM methods can be generally used to generate and execute various JavaScript statements.

4.5.3.5 Structural Analysis of Other Types of DJS

As mentioned before, the DJS instances generated by the `document.write` method() and the `innerHTML` property are identified from three different sources. We use *jscode* to present a DJS instance identified between a pair of `<script>` and `</script>` tags, use *eventhandler* to represent a DJS instance identified in an event handler, and use *jsprotocol* to represent a DJS instance identified in a `javascript:protocol` URL. The DJS instances generated by the DOM methods are specified in either the `src` attribute or the `text` attribute of a `script` element. Table 4.6 gives the structural analysis results of the DJS instances generated by these three dynamic generation techniques. The main usage of each type of DJS instance is summarized in the last column of the Table 4.6.

4.5.3.6 Safe Alternatives to `jscode` Generation via `document.write()` and `innerHTMLHTML`

For the eventhandler and `jsprotocol` `DJS` instances generated by `document.write()` and `innerHTMLHTML`, their usages are relatively safe. When new content is added to a document, event handlers are directly specified on various elements of the newly-added content to respond to various events. The `javascript:protocol` scripts are often used on links to execute some statements without loading a new document.

What we emphasize is that generating `jscode` using `document.write()` and `innerHTMLHTML` is not desirable. For `document.write()`, the generated `jscode` is immediately executed. Multiple `document.write()` calls can be used to construct a `jscode`, and `document.write()` calls can be nested. All these factors make the filtering of write-generated malicious JavaScript code a very challenging task [120]. However, our results show that 26,125 instances of write-generated `jscode` are identified on a large number of 4,000 homepages. For `innerHTMLHTML`, the generated `jscode` is recognized by a browser, but it is not necessarily executed. For example, Firefox does not directly execute a `jscode` generated by `innerHTMLHTML`. In Internet Explorer, the `defer` attribute and some tricks need to be used to execute an `innerHTMLHTML`-generated `jscode`, but this practice is also not recommended due to potential script-injection attacks [173]. Fortunately, only 503 instances of this practice are identified on 120 pages as shown in Table 4.6.

The best practice is to use DOM methods to dynamically generate JavaScript code. Using DOM methods (such as `createElement()` and `createTextNode()`) to create JavaScript elements explicitly declares that the new elements are scripts. This practice can enable

potential Web content protection mechanisms such as those presented in [51, 88, 120] to accurately define security policies and weed out potential malicious JavaScript code. Unfortunately, only 1,927(1,866 plus 61) instances of this practice are identified.

Our results show that the main usage of jscode generated by `document.write()` and `innerHTML` is for including other JavaScript files (denoted as *JS inclusion* in Table 4.6). Indeed, by specifying the `src` attribute of a script element, DOM methods fit well for such a usage. By specifying the `text` attribute of a script element, DOM methods can also be used to generate and execute various statements such as assignment statements or function calls, thus safely replacing the relatively insecure practices of jscode generation via `document.write()` and `innerHTML`.

4.5.4 Event Handler Registration

Our measurement results show that event handler registrations occurred on 6,451(94.9%) pages in the pre-onload phase, with an average of 108.2 registrations per page and a maximum of 5,074 registrations per page. Event handler registration occurred on 1,767(26.0%) pages in the post-onload phase, with an average of 61.4 registrations per page and a maximum of 2,229 registrations per page. These results include majority event types (e.g., event attributes of HTML tags, timer events, and XMLHttpRequest events) and event registration techniques supported in Firefox. The execution of event handlers may trigger further JavaScript inclusion and dynamic generation, implying that our captured insecure JavaScript practices are likely conservative estimates.

4.6 Related Work

To the best of our knowledge, there is no directly related work on characterizing the insecure practices of JavaScript inclusion and dynamic generation. Therefore, we only briefly review some JavaScript related measurement studies. Krishnamurthy and Wills [60] measured the homepages of 1,158 unique sites selected from Alexa.com [136] to study the content delivery tradeoffs in Web access. The focus of their study is on the performance impact of extraneous content, and their results show that JavaScript is often used on popular webpages to retrieve extraneous content such as images and advertisements.

In the investigation of malware, several execution-based measurement studies [79, 87, 107] have been conducted to identify malicious webpages that contain code (in many cases, JavaScript code) for exploiting Web browser vulnerabilities and installing malware. Instead of targeting at malicious sites, our focus in this work is on legitimate websites' insecure JavaScript practices.

4.7 Summary

In this work, we presented the first measurement study on insecure practices of using JavaScript on the Web. We focused on investigating the severity and nature of insecure JavaScript inclusion and dynamic generation. Through an instrumented Mozilla Firefox 2 Web browser, we visited the homepages of 6,805 popular websites in 15 different categories. We found that at least 66.4% of the measured websites have the insecure practices of including JavaScript files from external domains into the top-level documents of their homepages. Our in-depth analysis on the domain name relationship between JavaScript file inclusion

sites and hosting sites further reveals the severity and nature of those insecure practices. Our measurement results on JavaScript dynamic generation show that the “evil” function `eval()` was called on 44.4% of the measured homepages, and the `document.write()` method and the `innerHTML` property were also used to generate JavaScript code. Our AST-based structural analysis on various DJS instances further uncovers their usages with respect to programming language functionality. Our analysis indicates that in common cases, safe alternatives do exist for both the insecure JavaScript inclusion and insecure JavaScript dynamic generation. Since Web-based attacks have become more common and damaging in recent years, we suggest website developers and administrators pay serious attention to these insecure JavaScript practices and use safe alternatives to avoid them. In the future, we will measure insecure JavaScript practices on more specific types of websites and webpages. We will also investigate whether other insecure JavaScript practices exist on the Web.

Chapter 5

Secure and Convenient Kiosk

Browsing

In this chapter, we present our work on secure and convenient kiosk browsing. Many people use public computers provided in kiosk environments to browse the Web. Unfortunately, public computers are usually far less trustworthy than peoples' own computers. We present a simple approach that enables an extended browser on a mobile device and a regular browser on a public computer to collaboratively support a Web session. Using this approach, a user can securely perform sensitive interactions on the mobile device and conveniently perform other browsing interactions on the public computer.

5.1 Motivation

Web browsing has become such an integral part of our everyday lives that we use browsers to perform many important tasks such as banking, shopping, and bill-paying. To facilitate

ubiquitous Web access, many kiosk environments such as cafés, airport lounges, and hotel business centers provide people with Internet-connected public computers. These public computers often have high-speed network connections. They are also convenient to use since they normally have full-size keyboards and large displays. People who do not own a computer or carry a laptop with them frequently use these public computers to browse the Web.

Unfortunately, public computers are usually far less trustworthy than peoples' own computers. By "trustworthy", we mean that it is less likely that malware or spyware has been installed on a computer to log user input, steal account information, and even secretly hijack a secure (HTTPS) Web browsing session to make fraudulent transactions. Public computers are used by many people to run different applications and visit various websites; consequently, it is very likely for them to be infected with malware or spyware. Simply searching "public computer security" online, we can find numerous articles suggesting that people should not use public computers to perform sensitive activities. For example, Microsoft suggests that to be really safe, a user should not enter any sensitive information into a public computer [134].

To secure kiosk computing environments, researchers have proposed a number of solutions [18, 29, 36, 49, 70, 71, 72, 83, 84, 92, 98, 99, 113]. Most of these solutions use a trusted mobile device such as a PDA (Personal Digital Assistant) or a mobile phone to enhance the security of kiosk computing environments, and we refer to them as *PDA-based* solutions. Mobile devices are favored by PDA-based solutions because (1) they are more portable than desktop and laptop computers, and (2) they are generally more trustworthy than public computers. Nevertheless, using small user interfaces on mobile handheld devices

is inherently difficult [108].

Many of these PDA-based solutions focus on specific objectives such as securing application or data access [83, 99], securing user authentication or input [18, 70, 72, 84, 113], and verifying software integrity [36], so they cannot be easily adopted to secure an entire Web browsing session. Some solutions do have the objective of securing an entire kiosk browsing session [49, 71, 92, 98], but they suffer from a few drawbacks that limit their practical use.

In this work, we propose *SessionMagnifier*, a simple approach to secure and convenient kiosk browsing. *SessionMagnifier* also relies on a trusted mobile device and is a PDA-based solution. However, our position is that with the support of a *trusted mobile device* (referred to as *PDA*), a solution to the problem of securing Web browsing on an *untrusted public computer* (referred to as *PC*) should, in essence, strive to synthesize the usability advantages of a PC and the security advantages of a PDA. Otherwise, a user can simply take the security risks of only using a PC, or a user can simply tolerate the inconvenience of only using a PDA with its small keyboard and display. Note that we use the term PDA to represent either a mobile phone or a PDA in this chapter, and we expect people to eventually use our solution on mobile phones which are more popular than PDAs.

SessionMagnifier is designed as a browser extension, and the key idea is to enable an extended browser on a PDA and a regular browser on a PC to collaboratively support a Web session. After a user types in the address of a website and initiates a Web session from the PDA, the extended browser on the PDA accurately synchronizes a modified copy of its latest webpage document to a regular browser on the PC. The copied webpage document is modified to achieve accurate webpage rendering on the PC browser, to track a user's interaction with the same webpage on the PC browser, and to prevent sensitive information

from leaking to the PC.

This solution is simple and practical because a user only needs to carry a trusted mobile device and install a SessionMagnifier extension to the device's Web browser – no third-party proxy is needed, no installation or configuration on an untrusted computer is needed, no Web server modification is needed, and no extra cryptographic key exchange is needed. SessionMagnifier provides a strong security guarantee because end-to-end security is directly established between a trusted mobile device and a remote Web server; meanwhile, simple and explicit communication interfaces are defined to enforce strong isolation between a PDA and a PC. SessionMagnifier enables a user to fully take advantage of the convenience of using a PC. This is because only very sensitive interactions such as entering username and password need to be directly performed from the PDA while all other browsing interactions can be conveniently performed from the PC.

We implemented SessionMagnifier for Mozilla's Fennec mobile browser [171]. We installed Fennec and SessionMagnifier on a Nokia N810 Internet Tablet and conducted evaluations on usability, performance, and feasibility. Our evaluation and analysis demonstrate that the proposed simple solution can be practically used to support secure and convenient Web browsing on untrusted public computers.

The remainder of this chapter is structured as follows. Section 5.2 reviews related work. Section 5.3 details the design of SessionMagnifier. Section 5.4 describes the implementation of SessionMagnifier. Section 5.5 analyzes the security of SessionMagnifier. Section 5.6 presents the usability, performance, and feasibility evaluation of SessionMagnifier. Finally, Section 5.7 summarizes this work.

5.2 Related Work

Balfanz and Felten [5] introduced a *splitting-trust paradigm* to divide an application between a small trusted mobile device and a bigger, more powerful, but possibly untrusted computer. SessionMagnifier is inspired by this paradigm; however, we do not split a browser but instead enable an extended browser on a trusted mobile device and a regular browser on an untrusted computer to collaboratively support a Web session. The splitting-trust paradigm has also inspired many other kiosk computing solutions that rely on a trusted mobile device. We classify these solutions into four categories based on their different objectives.

5.2.1 Securing Application or Data Access

Oprea et al. [83] proposed a *three-party* secure remote terminal architecture to enable users to access their sensitive home computing environment via a trusted mobile device and an untrusted terminal. This three-party architecture is based on a thin-client VNC (Virtual Network Computing) remote display system [90], in which a VNC server can update the framebuffer displayed on a VNC client. Sharp et al. [99] proposed a VNC-based *thin-client* architecture to support secure access to unmodified applications. This architecture is similar to the three-party architecture [83], but it provides additional mechanisms to obfuscate the content displayed on an untrusted display. These VNC-based secure application or data access solutions work at the framebuffer level with high overhead, so they cannot be naturally adopted to support smooth Web interactions. In addition, trusted VNC servers must be deployed in these solutions.

5.2.2 Securing User Authentication or Input

Parno et al. [84] built a *Phoolproof* phishing prevention system that uses a trusted mobile device to perform mutual authentication between a user and a website. Mannan and Oorschot [70] proposed the *MP-Auth* protocol, in which a trusted mobile device turns a long-term password into a one-time password via the public key of an intended server; therefore, a user's long-term password will not be revealed to phishing sites or untrusted computers. McCune et al. [72] proposed a *BitE* framework that leverages the features of TPM (Trusted Platform Module) [187] to establish an encrypted input tunnel from a trusted mobile device to an application running on a TPM-equipped untrusted computer. Clarke et al. [18] and Wu et al. [113] designed protocols that rely on both a trusted third-party proxy and a trusted mobile device to secure authentication on untrusted computers. In addition, Florencio and Herley [29] proposed approaches to secure password input on untrusted computers without using mobile devices. All these solutions focus on securing user authentication or input, so they are not directly applicable for securing Web browsing sessions.

5.2.3 Verifying Software Integrity

Garriss et al. [36] built a system that uses a mobile device to establish trust in a kiosk computing environment. This system employs both a TPM [187] module equipped on a kiosk computer and an integrity attestation server of the kiosk, and it focuses on verifying the identity and integrity of software loaded on a public computer before revealing sensitive information to the computer. However, our SessionMagnifier focuses on securing Web browsing sessions on potentially untrusted public computers.

5.2.4 Securing Web Browsing Sessions

A few kiosk computing solutions share the same objective with our SessionMagnifier: securing Web browsing sessions. Ross et al. [92] proposed a *composable secure proxy* architecture to provide secure multi-modal access to Web services from any device. A similar proxy-based architecture called *Delegate* [49] was proposed to enable users to access Web services from untrusted computers. In these solutions, essentially it is the browser on an untrusted computer that accesses remote Web servers; meanwhile, secure proxies perform content and control filtering functionalities. Two main obstacles impede the adoption of these proxy-based solutions. First, secure third-party proxies must be widely deployed, well managed, and fully trusted by users. Second, to secure Web browsing, a proxy must use very complicated and comprehensive rules to validate requests, remove sensitive content, maintain user information, and manage session information such as HTTP cookies.

Margolin et al. [71] introduced a *Guardian* framework that uses a PDA as a proxy for all interactions between an untrusted computer and remote Web servers. Guardian uses LWP::UserAgent Perl module to interact with remote servers, and uses the HTTP::Daemon::SSL module to interact with the untrusted public computer. This framework eliminates the requirement of using secure third-party proxies by moving their content and control filtering functionalities to a PDA. However, since it is still the browser on the untrusted computer that accesses remote Web servers, this solution does not reduce the inherent complexity of proxy-based solutions. Our SessionMagnifier directly uses the Web browser on a trusted mobile device to access remote Web servers, so it provides strong security assurances and greatly reduces the complexity of content and control filtering.

Recently, Sharp et al. [98] proposed a *split-trust browsing* architecture to explore splitting trust at the HTML level for Web applications. However, this architecture has three drawbacks that limit its practical application. First, its critical component the RDC (Remote Device Communication) agent must be installed on an untrusted computer. Second, its end-to-end security between a trusted mobile device and a remote Web server depends on an extra authentication and key-exchange process coordinated by the RDC agent. Third, it assumes that either Web applications are explicitly written or secure HTML-rewriting proxies are used to support split-trust browsing. In contrast, our SessionMagnifier is much simpler and more practically applicable – nothing needs to be installed or configured on an untrusted computer, end-to-end security between a trusted mobile device and a remote Web server is ensured by existing HTTPS connections, no third-party proxy is needed, and no modification needs to be made to existing Web applications.

5.3 Design

In this section, we first use a motivating example to illustrate the use of SessionMagnifier in a kiosk browsing environment. We then define the threat model and assumptions under which SessionMagnifier operates. Finally, we present the architecture design of SessionMagnifier.

5.3.1 A Motivating Example

Alice goes on a trip without carrying her laptop, but she wants to bid an item at eBay.com. During the last hours of the bidding, she needs to check the latest bidding status and make appropriate adjustments as necessary. Alice takes a PDA (or a mobile phone) with her, but she feels uncomfortable to continuously use the small keyboard and display of the PDA. She

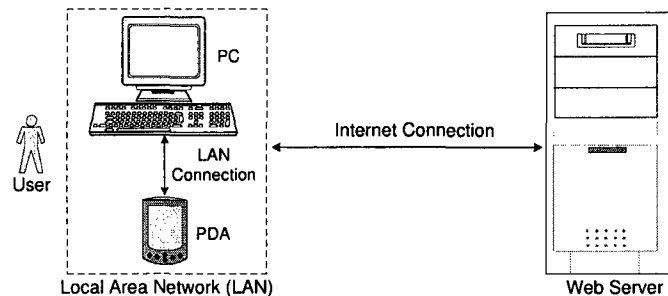


Figure 5.1: Kiosk browsing environment.

finds a public computer in an Internet café, but she has concerns about the security and privacy of using this public computer to log into her online accounts. Figure 5.1 illustrates such a kiosk browsing environment, and the problem here is how Alice can securely and conveniently sign into her online accounts and make transactions.

Fortunately, Alice can use a simple SessionMagnifier extension installed on the PDA browser to solve the problem. Alice first uses the PDA to connect to the Internet and signs into her eBay account from the PDA browser. The Internet connection is established either directly via Wi-Fi or indirectly via a USB-based or Bluetooth-based virtual network adapter of the public computer. Next, Alice turns on the SessionMagnifier extension installed on the PDA browser. She then types a URL address displayed by SessionMagnifier into the address bar of a regular PC browser and enables the connection between the PDA browser and the PC browser.

Starting from this point, SessionMagnifier synchronizes new webpage content from the PDA browser to the PC browser, and Alice can conveniently view and interact with the same webpage using the PC browser. Her interactions initiated from the PC browser will be sent back to SessionMagnifier and then securely sent out to eBay.com. Alice can verify and confirm any important interactions initiated from the PC browser, and she can also

use an “Auto On” toolbar button to bypass this verification and confirmation step for less important interactions. Meanwhile, using a “Sync On” toolbar button, Alice can switch on or off the synchronization on each specific webpage so that she can input and view sensitive information only on the PDA browser. Alice may continue the bidding process until she wins or loses the auction.

5.3.2 Threat Model and Assumptions

In a kiosk browsing environment, attackers are interested in stealing a user’s sensitive information such as username and password to commit identity theft. They are also interested in hijacking a user’s browsing session to generate fraudulent transactions [43]. More specifically, attackers may use the following five types of attacks to achieve their goals.

- **input stealing** – acquire sensitive input information by using software or hardware keyloggers.
- **output stealing** – acquire sensitive output information by using screen or window capture software.
- **session information stealing** – acquire sensitive session information such as HTTP cookies and session IDs through malware or spyware.
- **session hijacking** – (secretly) control a session and make fraudulent transactions through malware.
- **network attacks** – perform the above stealing and hijacking attacks at the network-level.

We define the capabilities of an attacker at two different levels: host-level and network-level. With host-level capabilities, an attacker can install malicious hardware and software

on a public computer and perform the first four types of attacks listed above. With network-level capabilities, an attacker can eavesdrop or tamper with network messages to perform passive or active network attacks.

We consider three typical types of Web sessions: pure HTTPS sessions, pure HTTP sessions, and hybrid sessions. In a pure HTTPS session, a Web server uses SSL/TLS cryptographic protocols to protect all important interactions with an authenticated user, and a user can also authenticate the Web server by inspecting its certificate. In a pure HTTP session, a Web server does not provide any transport layer security protection. In a hybrid session, a Web server uses SSL/TLS cryptographic protocols to protect the user authentication process, but it uses both HTTP and HTTPS to serve an authenticated user. Pure HTTPS sessions are supported by high-security institutions such as banks and credit card companies. Hybrid sessions are used by service providers such as Yahoo Mail. Pure HTTP sessions are used by websites that provide less sensitive services. For pure HTTPS sessions, we grant an attacker both the host-level and network-level capabilities. For hybrid and pure HTTP sessions, we only grant an attacker the host-level capabilities. Recently, solutions such as SessionLock [2] and ForceHTTPS [42] have been proposed to enhance the security of hybrid sessions in potentially hostile network environments. For hybrid sessions, assuming the success of these solutions, we may further grant an attacker the network-level capabilities.

Like previous studies, we assume that a user's mobile device is a priori secure. In our design, the simple and explicit communication interfaces between a PDA and a PC further protect the security of the PDA. Given the prevalence of phishing attacks [20, 25], we also assume that a user is security conscious and is able to discern phishing through, for example,

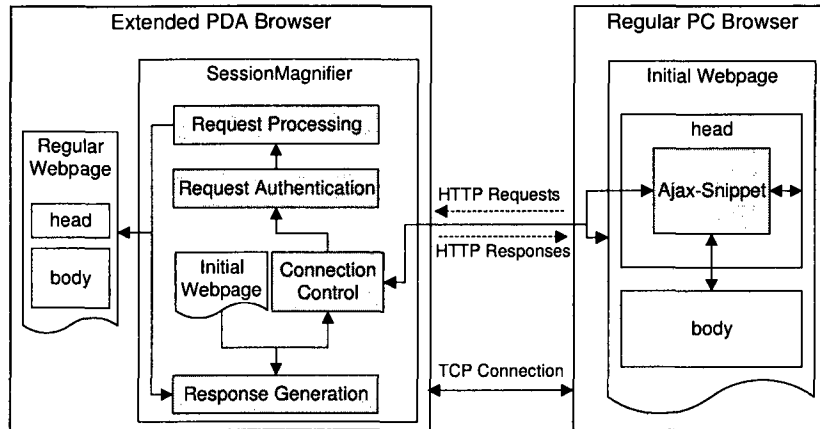


Figure 5.2: SessionMagnifier high-level architecture.

inspecting a Web server's certificate validated by the PDA browser.

5.3.3 Architecture Design

Figure 5.2 illustrates the high-level architecture of SessionMagnifier. A user simply installs the SessionMagnifier extension on a PDA browser; nothing needs to be installed or configured on a regular PC browser, and no third-party proxy is required. At the network layer, the PC can access the PDA via TCP connections. At the application layer, the regular PC browser communicates with the extended PDA browser using the HTTP protocol. A user directly uses the PDA browser to establish a Web session with a remote Web server. The SessionMagnifier extension is responsible for synchronizing the latest HTML webpage document from the PDA browser to the PC browser, and it is also responsible for accepting interactions initiated from the PC browser and securely performing these interactions on the PDA browser.

The simple architecture of SessionMagnifier leverages two important features of modern Web browsers: end-user extensibility [147, 157] and Ajax (Asynchronous JavaScript

and XML) technology [135]. End-user extensibility allows the SessionMagnifier browser extension to maximize its capabilities and seamlessly integrate its functionalities with modern browsers. Ajax technology enables a regular PC browser to periodically send HTTP requests to SessionMagnifier and maintain the communication with the PDA browser. End-user extensibility is well supported by popular browsers such as Firefox [147] and Internet Explorer [157], and Ajax technology has received wide acceptance among all popular Web browsers [135]. Therefore, SessionMagnifier can be practically implemented and deployed on popular Web browsers.

In a kiosk browsing environment, establishing TCP connections between a PDA and a PC is feasible, and having Internet access for a PDA is also feasible. Using Wi-Fi, a user can easily establish both types of network connections. If Wi-Fi is not available, a user can use USB or Bluetooth to enable TCP connections between a PDA and a PC; meanwhile, using various *Internet access over USB* or *Internet access over Bluetooth* techniques (e.g., Microsoft ActiveSync [168]), a user can also easily obtain Internet access for a PDA. Therefore, SessionMagnifier can be practically used in kiosks.

The SessionMagnifier extension consists of four main components: *connection control*, *request authentication*, *request processing*, and *response generation*. In addition, it also contains an *initial webpage*, which is an HTML file to be sent to a regular PC browser. We still use the previous motivating example to describe the roles played by the four components and the initial webpage in a kiosk browsing session.

5.3.3.1 Connection Control

When Alice turns on the SessionMagnifier extension installed on the PDA browser, the connection control component starts to work. This component uses a server socket to listen for new incoming connections from a PC. The server socket is TCP-based so that connections can be directly made from a regular PC browser. After the server socket binds to the IP address (e.g., 192.168.1.3, assigned by the kiosk LAN network) and a TCP port (e.g., 3000) of the PDA, the connection control component will display the URL address of SessionMagnifier (e.g., `http://192.168.1.3:3000`) to Alice.

Establishing the connection from a PC browser to the PDA browser is just like visiting a regular website. Alice simply types the URL address of SessionMagnifier into the address bar of the regular PC browser and sends out an initial HTTP request. When the connection control component of SessionMagnifier receives this initial HTTP request, it displays the source IP address of the request in a dialog box and asks Alice to confirm this connection. This confirmation dialog box is employed to help Alice make sure that the initial connection request does come from her PC.

If Alice accepts this initial connection request, the connection control component will read the initial webpage and send it to the PC browser. In an analogy, the initial webpage of SessionMagnifier is like the homepage of a regular website. The *body* of the initial webpage is very simple. It provides a simple form to ask Alice to submit a one-time password. The *head* of the initial page mainly contains Ajax-Snippet, which is a set of XHR (XMLHttpRequest) [135] related objects and functions. After the initial webpage is loaded on the PC browser, Ajax-Snippet will periodically send out “POST” type XHR

polling requests to the connection control component of SessionMagnifier. Ajax-Snippet sends an XHR polling request and the connection control component returns a response; therefore, all further communication between the PC browser and the PDA browser can be automatically carried out.

If Ajax-Snippet receives a response message that contains a new webpage document, it will smoothly update the head and body of the initial webpage to keep the webpage content on the PC browser synchronized with that on the PDA browser. Meanwhile, Ajax-Snippet always resides in the head of the current webpage on the PC browser to maintain the communication with the PDA browser. Ajax-Snippet uses “POST” type XHR polling requests so that any interaction information such as link clicking or form filling on the PC browser can be directly piggybacked onto an XHR polling request and sent to the PDA browser.

5.3.3.2 Request Authentication

The one-time password mentioned above is generated and stored by SessionMagnifier on the PDA browser. On the PC browser, the same password submitted by Alice will not be transmitted to the PDA; it is just stored and used by Ajax-Snippet to compute the HMAC (keyed-Hash Message Authentication Code) [153] for each XHR polling request. Before sending an XHR polling request, Ajax-Snippet computes an HMAC for the header and content of the request and appends the HMAC as an additional parameter of the request-URI [27].

When the connection control component of SessionMagnifier receives an XHR polling request, it will forward the request to the request authentication component. The request

authentication component will then compute a new HMAC for the received request (discarding the additional HMAC parameter) and compare the computed HMAC with the HMAC embedded in the request-URI. If the two HMACs are identical, the XHR polling request is regarded as valid and is further forwarded to the request processing component. We use such a request authentication mechanism to protect the browsing session on the PDA and to ensure that SessionMagnifier only processes the requests sent from Alice's PC browser.

5.3.3.3 Request Processing

When the request processing component receives a valid XHR polling request, it will perform two tasks: new content checking and interaction information merging. The former is to check whether new webpage document on the PDA browser needs to be synchronized to the PC browser. The later is to check whether a user's interaction information on the PC browser needs to be merged to the PDA browser.

SessionMagnifier keeps a timestamp for the latest webpage document on the PDA browser. A timestamp used by SessionMagnifier is the number of milliseconds since midnight of January 1st, 1970. Whenever a new webpage document is synchronized to the PC browser, the timestamp of the document is also sent to Ajax-Snippet using the same response message. Whenever Ajax-Snippet sends an XHR polling request to SessionMagnifier, it carries back the timestamp of the current webpage document on the PC browser using the same request message.

The request processing component compares the two timestamp values to determine whether the webpage document on the PC browser is outdated. If the timestamp of the webpage document on the PC browser is older than that on the PDA browser, the request

processing component informs the response generation component to synchronize the new webpage document on the PDA browser to the PC browser. Otherwise, it simply informs the response generation component to send back an empty response message to Ajax-Snippet.

The request processing component also examines the content of this “POST ” type XHR polling request to see whether any interaction information is carried back from the PC browser. If new interaction information is carried in the XHR polling request, the request processing component will further execute the following four steps to merge the interaction information to the PDA browser. First, it will accurately reflect the interaction information (e.g., form filling information) to the corresponding webpage elements on the PDA browser. Second, it will highlight these webpage elements and scroll them into the view window of the PDA browser. Third, it will display a modal dialog box to ask Alice to verify the highlighted webpage elements. Finally, if Alice confirms that the interaction information reflected on the PDA browser is what she did on the PC browser, the request processing component will actually perform the interaction (e.g., submitting a form) on the PDA browser; otherwise, the request processing component will undo the changes made in the first two steps and ignore the interaction information carried in this XHR polling request. By only performing confirmed interaction information on the PDA browser, the request processing component assures a user that important interaction information is not tampered with or injected by attackers.

5.3.3.4 Response Generation

The response generation component is the most critical component of SessionMagnifier, and it poses three main design challenges: (1) how to enable high-quality webpage document

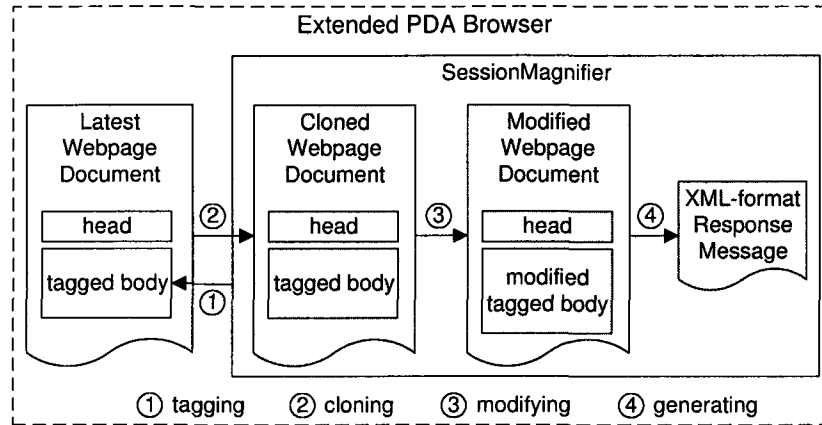


Figure 5.3: Response generation procedure.

synchronization from the PDA browser to the PC browser, (2) how to enable accurate user interaction on the PC browser, and (3) how to prevent sensitive information from leaking out of the PDA. When a new webpage document is loaded on the PDA browser, the response generation component uses the procedure shown in Figure 5.3 to generate a response message for the PC browser. This procedure consists of four main steps: *tagging*, *cloning*, *modifying*, and *generating*. We now detail these four steps to explain how we addressed the design challenges of the response generation component.

(1) *Tagging*: In the tagging step, the response generation component adds `id` attributes [154] to the *interested actionable elements* of the latest webpage document on the PDA browser. We define interested actionable elements as the elements on which keyboard or mouse interactions will trigger the loading of a new webpage document. For example, links, forms, and clickable input elements outside of the forms are all regarded as interested actionable elements.

Tagging allows SessionMagnifier to simply use unique element identifiers to accurately track interested actionable elements on both the PC browser and the PDA browser. By

directly tagging the webpage document on the PDA browser, the response generation component saves memory space and avoids the complexity of using any additional mapping mechanisms. Note that the response generation component only tags those interested actionable elements that do not have an `id` attribute, so it will not affect the behavior of the webpage document on the PDA browser.

(2) *Cloning*: In the cloning step, the response generation component uses the standard `cloneNode` DOM (Document Object Model) [191] method to clone a complete copy of the above tagged webpage document. Using a cloned webpage document has two advantages. One is that standard DOM methods can still be handily used to modify the webpage. The other is that any further modification is only made to this cloned copy without polluting the webpage document on the PDA browser.

(3) *Modifying*: In the modifying step, the response generation component makes three main modifications to the cloned webpage document: URL address modification, event handler modification, and sensitive information filtering.

In general, each HTML webpage document has a set of associated supplementary objects such as stylesheets, images, and scripts. After loading a webpage document synchronized from the PDA browser, the PC browser must also download the associated supplementary objects in order to accurately render the webpage. To support the downloading of supplementary objects, the response generation component changes all the relative URL addresses contained in the cloned webpage document to the absolute URL addresses of the original Web servers. Without such a modification, the PC browser will send all relative URL requests to the PDA browser because it actually always connects to the SessionMagnifier browser extension on the PDA browser.

To track Alice's interaction with the same webpage on the PC browser, the response generation component must change the event handlers of those interested actionable elements. For form elements, the response generation component changes their *onsubmit* event handlers by adding a call to a specific JavaScript function residing in Ajax-Snippet. Therefore, later on when Alice submits a form on the PC browser, the *id* attribute value and element values of the form will be passed to Ajax-Snippet and then sent back to SessionMagnifier via an XHR polling request. In a similar way, the response generation component changes *onclick* event handlers of links and other clickable input elements outside of the forms to track click interactions performed on the PC browser.

Filtering sensitive information is much simpler in SessionMagnifier than in existing solutions [49, 71, 92, 98]. The main reason is that SessionMagnifier only synchronizes webpage documents to the PC browser, and no session control information such as HTTP cookies will be leaked to the PC. Therefore, the response generation component only needs to filter out sensitive information contained in a webpage document itself. To achieve this goal, the response generation component mainly uses the following two strategies.

One is to remove any possibly sensitive information that is useless to the interaction and display of the webpage on the PC browser. For example, webpages often contain sensitive information such as session IDs in their URL links and form *action* attributes. The response generation component simply sets all form *action* attribute values to empty, and it also sets all link *href* attribute values to empty. Note that tracking form submitting and link clicking is enabled by the above event handler modifications; therefore, the original *action* and *href* attribute values are useless to the interaction of a webpage on the PC browser.

The second strategy is to obfuscate personalized sensitive information. The basic idea

is to replace user-specified sensitive information with information that is meaningless to attackers. For example, many websites display username information on their webpages for a logged-in user. Preventing the leakage of username information is important for protecting against attacks such as invasive advertising, password guessing, and even denial-of-service [12, 32]. SessionMagnifier maintains a rule table, in which simple filtering rules are defined by a user to specify which information should be obfuscated for each specific website. These rules could be defined for stable values (e.g., username), and they could also be defined for dynamic values (e.g., online banking balance) if the corresponding HTML elements of those values have stable IDs. The response generation component simply applies the rules to remove sensitive information contained in the cloned webpage document.

(4) *Generating*: In the last step, the response generation component extracts information from the modified webpage document and generates an XML-format response message. The response message is in XML-format so that later Ajax-Snippet can accurately extract structured response information from the *responseXML* attribute of an XHR object. The modified webpage document is an HTML document, but an XHR object expects to receive a valid XML document. Since HTML webpages are often malformed, directly sending a modified webpage document to Ajax-Snippet will often result in parsing errors. Therefore, the response generation component will extract essential head and body information from a modified webpage document, encapsulate the extracted information in an XML-format response message, and finally send out the response message to Ajax-Snippet.

5.3.3.5 Initial Webpage

We mentioned that after the initial webpage is loaded on the PC browser, Ajax-Snippet will always keep itself within the head of the current webpage on the PC browser and periodically send out “POST” type XHR polling requests to communicate with SessionMagnifier.

Whenever Ajax-Snippet receives an XML-format response message that contains a new webpage document, it will first use the head information contained in the response message to replace the head of the current webpage on the PC browser. To support proper rendering on different browsers such as Internet Explorer and Firefox, Ajax-Snippet detects the type of the PC browser and performs this replacement for each top-level child of the head element. Ajax-Snippet will then use the body information contained in the response message to replace the body of the current webpage on the PC browser. By combining the structural advantages of using DOM methods and the simplicity advantages of using the *innerHTML* property of HTML elements, Ajax-Snippet can smoothly and accurately keep the webpage content on the PC browser synchronized with that on the PDA browser.

Meanwhile, whenever Alice interacts with an interested actionable element of the synchronized webpage on the PC browser, a call to a specific JavaScript function is made to extract the interaction information. The extracted information will be carried in the content of the next XHR polling request and synchronized to SessionMagnifier.

5.4 Implementation

The SessionMagnifier browser extension is designed to be implementable on different Web browsers. Indeed, only the connection control component is browser-specific; the request

authentication, request processing, and response generation components and the initial webpage can all be implemented using standard JavaScript and HTML that are supported by modern Web browsers.

We implemented a full-fledged SessionMagnifier browser extension for Mozilla's Fennec browser [171], which is the mobile version of Firefox and is currently in alpha release. Similar to Firefox, Fennec provides full support for add-ons and rich Internet applications. Our SessionMagnifier extension for Fennec is purely written in JavaScript and HTML. Due to the space limit, we only briefly describe the implementation of the connection control component that is specific to Fennec, and delineate the webpage content and interaction synchronization capabilities of SessionMagnifier in our current implementation.

We implemented the connection control component of SessionMagnifier as a server socket object of Mozilla's nsIServerSocket interface [172]. For this server socket object, we created a socket listener object of Mozilla's nsIServerSocketListener interface [172] to asynchronously accept incoming TCP connections, and we also associated a data listener object of Mozilla's nsIStreamListener interface [172] to a connected socket transport to asynchronously accept HTTP requests. In a Fennec browser extension, these objects can be easily created and manipulated using JavaScript code to realize the functionality of the connection control component. Our preliminary investigation indicates that Internet Explorer also supports the creation of a server socket through its BHO (Browser Helper Object) extension mechanism [157].

For webpage content synchronization, SessionMagnifier supports various webpages including dynamic webpages (e.g., Google Maps) that use Ajax, CSS, or other DHTML techniques. SessionMagnifier detects dynamic webpage changes on a PDA browser and

synchronizes the new content to a PC browser. In principle, any type of webpage content could be synchronized by SessionMagnifier. However, the current version of SessionMagnifier cannot properly synchronize some webpages such as Gmail webpages due to the unfinished implementation on iframe elements. For interaction synchronization, our current implementation supports interactions on those *interested actionable elements* as defined in our design. However, SessionMagnifier can also easily synchronize any other interactions (e.g., those altering document elements without calling for new data) as long as their corresponding HTML elements support event handlers (which could be *onsubmit*, *onclick*, or any other handlers). We will provide support for other necessary interactions in our future implementation.

5.5 Security Analysis

The security assurances provided by SessionMagnifier can be attributed to three factors: using a trusted PDA, accessing a remote Web server directly from a PDA browser, and enforcing strong isolation between a PDA and a PC. We now analyze the security of SessionMagnifier based on the threat model and assumptions defined in our design. We must emphasize that SessionMagnifier aims to enhance the security of using an untrusted public computer for Web browsing, but it does not attempt to secure a kiosk environment itself. In other words, the security upper-bound of using SessionMagnifier is equivalent to that of using a user's own laptop computer in a kiosk environment.

Using a trusted PDA, SessionMagnifier is robust against input stealing attacks and output stealing attacks. A user simply enters sensitive information such as username and

password on the PDA browser, so the keyloggers installed on the PC cannot acquire sensitive input information. Meanwhile, user-specified information is obfuscated by the response generation component of SessionMagnifier, so it is very hard for screen or window capture software installed on the PC to acquire sensitive output information. The ability of SessionMagnifier to provide these two types of security assurances is the same as that of other solutions to securing kiosk browsing sessions [49, 71, 92, 98].

Accessing a remote Web server directly from the PDA browser is a unique feature of SessionMagnifier because other solutions [49, 71, 92, 98] all use the browser on an untrusted computer to establish a Web session with a remote Web server. Besides, SessionMagnifier enforces a strong isolation between a PDA and a PC by only allowing HTTP communications. Combining these two factors, SessionMagnifier provides high security assurances to protect against other three types of attacks: session information stealing attacks, session hijacking attacks, and network attacks.

SessionMagnifier is robust against session information stealing attacks. Since SessionMagnifier only synchronizes the content of a modified HTML document from the PDA browser to the PC browser, session information such as HTTP cookies will never be leaked to the PC. Meanwhile, since all useless values such as form *action* attribute values and link *href* attribute values are simply set to empty by the response generation component of SessionMagnifier, no session IDs contained in these values will be revealed to the PC. Preventing the leakage of HTTP cookies and session IDs is important because an attacker can use them to further steal other sensitive user data or hijack browsing sessions. Unfortunately, this type of security assurance is not considered in [98], and it is considered in [49, 71, 92] by employing very complex filtering rules and mapping mechanisms.

SessionMagnifier is robust against session hijacking attacks. Since a Web session is established between the PDA browser and a remote Web server, malware installed on the PC cannot directly seize the control of a session to make fraudulent transactions. The only possible way for an attacker to hijack a session is to tamper with or inject interaction information in an XHR polling request. SessionMagnifier defends against such attacks by using its request processing component to accurately reflect interaction information on the PDA browser and only perform user-confirmed interactions. Protection against session hijacking is also considered in [49, 71, 92, 98]. However, because these solutions use the PC browser to establish a Web session with a remote Web server, they must perform very complex request validations but still cannot achieve the same security level as that of SessionMagnifier.

In terms of network attacks, SessionMagnifier ensures end-to-end security between a trusted mobile device and a remote Web server by directly using existing HTTPS connections. For pure HTTPS sessions, SessionMagnifier is robust against network attacks. For hybrid sessions, SessionMagnifier is robust against network attacks for HTTPS webpages. For pure HTTP sessions, there is no strong incentive to defend against network attacks because in general Web servers that do not use SSL/TLS only provide less sensitive services. Existing solutions [49, 71, 92, 98] provide similar security guarantees against network attacks as provided by SessionMagnifier; however, they often necessitate an additional SSL/TLS connection or encryption channel still mainly because they use the PC browser to establish a Web session with a remote Web server.

5.6 Evaluation

In this section, we focus on evaluating the usability of SessionMagnifier. We also briefly present the performance and feasibility evaluation results.

5.6.1 Usability Evaluation

Our primary goal is to measure whether using SessionMagnifier is more convenient than merely using a PDA browser. To achieve this goal, we conducted a usability study based on a real eBay bidding scenario.

5.6.1.1 Participants

Twenty-two adults, 11 females and 11 males, participated in this study. Nineteen participants were between ages of 18 and 30, and three participants were over 30 years old. We did not screen participants based on experiences using different Web browsers, using mobile devices, or using eBay services.

5.6.1.2 Scenario and Procedure

We presented such a scenario to each participant: “Suppose you want to bid a book titled *‘Xbox 360 games in a nutshell’* at eBay.com. You visit www.ebay.com and sign into an eBay testing account. You search the book using its title and find the item. You place a higher bid by adding one dollar and get confirmation that you are currently the highest bidder. Finally you sign out of the eBay website.”. Note that the book item was added to eBay using a seller’s account created by us, and the eBay testing account was also created by us.

TA1: On PDA, type the address www.ebay.com into the address bar of Fennec
TA2: On the “homepage” of eBay, click on the “Sign in” link
TA3: On the “welcome page” of eBay, sign into the testing account
TA4: On the “logged page”, type “Xbox 360 games in a nutshell” into the “Find” input field
TA5: On the “logged page”, click on the “Search” button
TA6: On the “search result page”, click on the link item “Xbox 360 games in a nutshell”
TA7: On the “item page”, type a higher amount into the “Your maximum bid” input field
TA8: On the “item page”, click on the “Place Bid” button
TA9: On the “review and confirm bid page”, click on the “Confirm Bid” button
TA10: On the “bid confirmation page”, click on the “Sign out” link

Table 5.1: The 10 tasks in procedure A.

We asked each participant to perform this eBay bidding scenario using two procedures A and B. In procedure A, a participant only uses a PDA; in procedure B, a participant uses both a PDA and a PC. We used a Nokia N810 Internet Tablet as the PDA, and we pre-installed a Fennec browser [171] and our SessionMagnifier browser extension on it. In procedure A, each participant used the Fennec browser (with SessionMagnifier turned off) on the PDA to perform the bidding scenario. In procedure B, each participant used the Fennec browser (with SessionMagnifier turned on) on the PDA and a regular browser on a PC to perform the bidding scenario. We randomly assigned 11 participants to first perform procedure A and the other 11 participants to first perform procedure B. Before a test, we trained each participant on the use of the PDA and the Fennec browser. We also explained the purpose of SessionMagnifier, and it seems that all the participants understood the threats addressed by SessionMagnifier.

The step-by-step tasks for the two procedures are listed in Tables 5.1 and 5.2, respectively, and we presented these two task lists to each participant. Procedure A has 10 tasks and procedure B has 18 tasks. Each task is a specific browsing action such as clicking on the “Sign in” link or typing “Xbox 360 games in a nutshell” into the “Find” input field.

TB1: On PDA, type the address www.ebay.com into the address bar of Fennec
TB2: On PDA, on the “homepage” of eBay, click on the “Sign in” link
TB3: On PDA, on the “welcome page” of eBay, sign into the testing account
TB4: On PDA, click on the “Sync On” toolbar button
TB5: On PC, type the address http://192.168.1.3:3000 into the address bar of a browser
TB6: On PC, submit the one-time password to synchronize browsers
TB7: On PC, on the “logged page”, type “Xbox 360 games in a nutshell” into the “Find” input field
TB8: On PC, on the “logged page”, click on the “Search” button
TB9: On PDA, verify the highlighted form on the “logged page” and confirm the form submission action
TB10: On PC, on the “search result page”, click on the link item “Xbox 360 games in a nutshell”
TB11: On PDA, verify the highlighted link on the “search result page” and confirm the click action
TB12: On PC, on the “item page”, type a higher amount into the “Your maximum bid” input field
TB13: On PC, on the “item page”, click on the “Place Bid” button
TB14: On PDA, verify the highlighted input field on the “item page” and confirm the input action
TB15: On PC, on the “review and confirm bid page”, click on the “Confirm Bid” button
TB16: On PDA, verify the highlighted button on the “review and confirm bid page” and confirm the click action
TB17: On PC, on the “bid confirmation page”, click on the “Sign out” link
TB18: On PDA, verify the highlighted link on the “bid confirmation page” and confirm the click action

Table 5.2: The 18 tasks in procedure B.

Procedure B has more tasks because we asked each participant to verify and confirm all the interaction information sent back to the PDA browser. For example, Figure 5.4 shows a picture of performing the task TB9 on Fennec. After the search of “Xbox 360 games in a nutshell” performed on the PC browser is reflected on Fennec, SessionMagnifier highlights the border of the search form with red color and displays a modal dialog box on the PDA. Using this dialog box, a participant can either confirm this form submission by clicking on the “OK” button or ignore this form submission by clicking on the “Cancel” button.

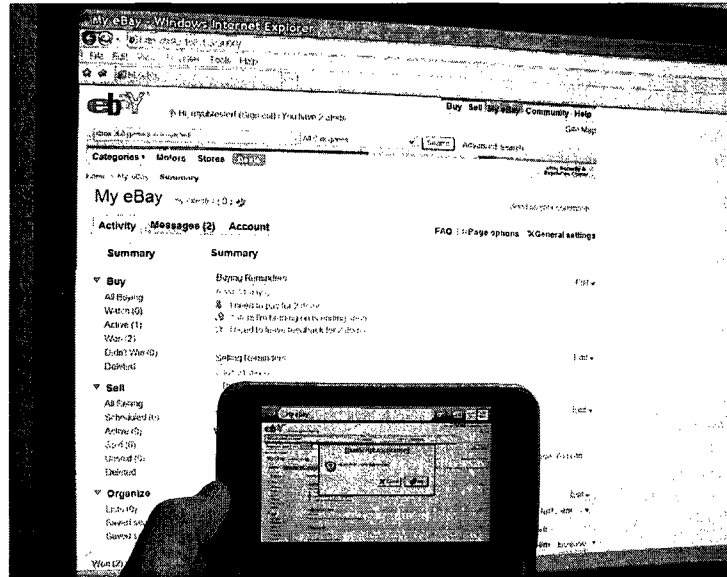


Figure 5.4: Performing the task TB9 on Fennec.

5.6.1.3 Data Collection

We collected data through observation and obtaining feedback from participants. When a participant was performing the two procedures, we observed the progress of the tasks. After a participant finished the two procedures, we asked the participant to answer 16 close-ended five-point Likert-scale (Strongly disagree, Disagree, Neither agree nor disagree, Agree, Strongly Agree) [166] questions. These 16 questions are listed in Table 5.3 (Q1 to Q6 were asked for both procedures). We also asked participants to write down open-ended comments on using SessionMagnifier.

5.6.1.4 Results and Analysis

We observed that all the 22 participants successfully finished the two procedures. We converted the responses to the Likert-scale questions to numeric values (1=Strongly disagree,

The six questions common to both procedures A and B (replacing the 'X' in the questions with 'A' or 'B')
Q1: Typing into input fields of a webpage in procedure X is easy
Q2: Clicking links of a webpage in procedure X is easy
Q3: Clicking buttons of a webpage in procedure X is easy
Q4: Scrolling a webpage in procedure X is easy
Q5: Viewing webpage content in procedure X is easy
Q6: Overall, performing procedure X is easy
The four questions specific to procedure B
QB1: Typing the URL address http://192.168.1.3:3000 of SessionMagnifier into the address bar of a PC browser in procedure B is easy
QB2: Clicking the "Sync On" toolbar button in procedure B is easy
QB3: Verifying a highlighted element (form, link, button) in procedure B is easy
QB4: Confirming an action (form, link, button) using the dialog box in procedure B is easy

Table 5.3: The 16 close-ended questions.

2=Disagree, 3=Neither agree nor disagree, 4=Agree, 5=Strongly Agree) and compared the responses to procedures A and B using t-tests. Strictly speaking, since the responses are ordinal data, they do not necessarily have interval scales. However, in practice this type of analysis is acceptable [16].

Figure 5.5 illustrates the mean ratings to questions Q1 to Q6 for the two procedures. We can see that for all the six questions the mean ratings to procedure B are much higher than those to procedure A. The t-tests (with 95% confidence interval) further reveal that the mean rating differences between the two procedures are significant for each of the six questions. These results clearly indicate that SessionMagnifier enables users to exploit the usability advantages of using the large keyboard and display of a PC.

Using a similar method, we analyzed the responses to the four questions specific to procedure B. The mean ratings to questions QB1 to QB4 are: 3.96, 4.09, 3.64, and 3.86, respectively. One-sample t-test (with 95% confidence interval) against the test value of three shows that the mean ratings to these questions are higher than three with statistical significance. These results indicate that performing the specific interactions introduced in

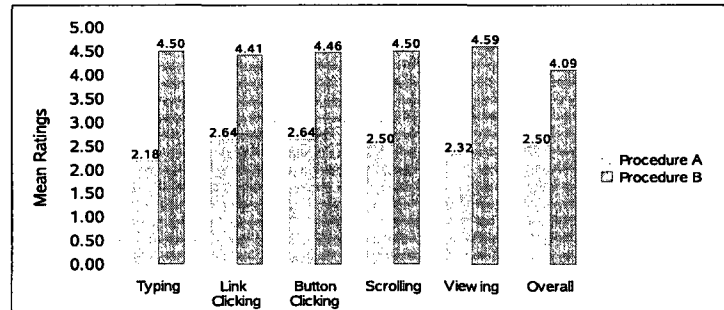


Figure 5.5: Mean ratings to questions Q1 to Q6.

procedure B is not difficult to users.

We further analyzed participants' open-ended comments on using SessionMagnifier. We found that 14 participants clearly mentioned that typing into input fields and viewing webpages are very convenient in procedure B. We also found that nine participants mentioned that it would be better if the number of verifying and confirming steps could be reduced. We should note that verification and confirmation are necessary steps for important interactions, and they were or should be considered in other splitting-trust based solutions. Moreover, SessionMagnifier allows a user to bypass this verification and confirmation step using the "Auto On" toolbar button. In procedure B, we disabled the "Auto On" feature to measure the worst case usability, but we believe that a user can actually be trained to use this feature to confidently bypass less important interactions.

5.6.2 Performance and Feasibility Evaluation

In our performance evaluation, we mainly measured the speed of SessionMagnifier in response generation (i.e., the procedure illustrated in Figure 5.3) and response transmission. We used Fennec to visit five homepages. The page size and response generation time (average of five runs) of these homepages are listed in Table 5.4. We can see that the larger

Site Name	Page Size (KB)	Generation Time (second)
google.com	8.9	0.36
ebay.com	49.5	1.37
bestbuy.com	80.2	2.64
weather.com	148.7	2.71
amazon.com	201.1	3.03

Table 5.4: Page size and response generation time of five homepages.

and more complex the HTML page is, the more generation time is needed. Response generation is not very efficient for large webpages, but we believe that the main reason is the poor memory management in the current alpha release of Fennec. Using the Linux `top` command, we observed that even without SessionMagnifier, Fennec requires over 107% of memory (128MB RAM) on Nokia N810 Internet Tablet by just loading the google.com homepage; however, the built-in browser on Nokia N810 only requires less than 78% of memory even when loading the amazon.com homepage. We believe that increasing memory or an improved Fennec can help reduce the response generation time of SessionMagnifier. In terms of the response transmission speed, since the PDA and the PC are located in the same LAN, the generated response message can normally be transmitted from the PDA to the PC within a second.

In our feasibility evaluation, we mainly tested whether TCP connections between a PDA and a PC can be established via Wi-Fi in kiosk environments. We conducted experiments at 20 public places (seven hotels, seven restaurants, three libraries, two gyms, and one coffee shop) that offer free Wi-Fi Internet access. Since some places do not provide public computers, we used a laptop to act as a public PC. At each place, we did not do any special configuration on either the PDA (the Nokia N810 Internet Tablet) or the PC, but just connected them to the same Wi-Fi access point to acquire IP addresses. We observed that TCP connections between the PDA and the PC are blocked (perhaps due to strict

security restrictions) at three hotels and two restaurants. At the other fifteen places, the PC can connect to the PDA using a TCP port (e.g. 3000), and we successfully performed Web browsing using SessionMagnifier. These results indicate that it is practical to use SessionMagnifier at many free Wi-Fi hotspots. Meanwhile, as mentioned in the design section, a kiosk environment that plans to enable SessionMagnifier can also use USB or Bluetooth, in addition to Wi-Fi.

5.7 Summary

We presented SessionMagnifier, a simple approach to secure and convenient kiosk browsing. Leveraging the end-user extensibility and Ajax technology of modern Web browsers, SessionMagnifier enables an extended browser on a mobile device and a regular browser on a public computer to collaboratively support a Web session. SessionMagnifier strives to synthesize the usability advantages of a public computer and the security advantages of a mobile device. Since a Web session is directly established between the PDA browser and a remote Web server, SessionMagnifier provides a strong end-to-end security guarantee and greatly reduces the complexity of content and control filtering. Since a user can perform the majority of browsing interactions from the PC and only perform very sensitive interactions from the PDA, SessionMagnifier enables a user to fully take advantage of the convenience of using a PC. We presented the design of SessionMagnifier in detail and analyzed the security of SessionMagnifier using a rigorous threat model. We implemented SessionMagnifier for Mozilla's Fennec browser and evaluated its usability, performance, and feasibility. Our evaluation and analysis demonstrate that SessionMagnifier is simple, secure, and usable.

In future work, we will enhance the implementation and evaluation of SessionMagnifier. In particular, we will improve our usability evaluation, for example, by gathering information about participants' experience with mobile devices, by allowing participants to use our tool with only basic instead of step-by-step instructions, by collecting data illustrating participants' thoughts on security aspects of using our tool, and by incorporating some security attack scenarios.

Chapter 6

Conclusions and Future Work

In the previous chapters, we have introduced this dissertation and detailed its four main contributions in enhancing Web browsing security. In this final chapter, we make conclusions and outline the future work.

6.1 Conclusions

Traditionally, attackers mainly focused on directly breaking into Web servers to acquire sensitive information or damage the systems. However, in recent years, Web browsers have gained increasing popularity as new attack vectors for launching various severe attacks such as drive-by download, cross-site scripting, phishing, privacy, and even large-scale denial of service attacks. Browser-based attacks are rampant over the Internet and have caused substantial damage to both Web users and service providers. Enhancing Web browsing security is therefore of great need and importance, and it has also become much more challenging than before.

This dissertation concentrates on enhancing the Web browsing security through exploring and experimenting with new approaches and software systems. Specifically, we have systematically studied four challenging Web browsing security problems: HTTP cookie management, phishing, insecure JavaScript practices, and browsing on untrusted public computers. We have proposed new approaches to address these problems, and built unique systems to validate our approaches. Our contributions include an automatic HTTP cookie management system, a transparent phishing protection system, a large-scale measurement of insecure JavaScript practices on the Web, and a simple approach to secure and convenient kiosk browsing. The four Web browsing security problems studied in this dissertation are real-world important problems, and our solutions presented in this dissertation are unique, meaningful, and useful.

6.2 Future Work

As malicious activities have increasingly become Web-based and as attackers have become more and more economically motivated and adaptable, enhancing Web browsing security will continue to be of critical importance. In the near future, I will keep the focus of my research on security, especially Web browsing security. For one example, based on the insecure JavaScript practice characterization work presented in Chapter 4, I will study JavaScript obfuscation and de-obfuscation activities to build browser-based anomaly intrusion detection systems. For another example, based on my experience in building CookiePicker (Chapter 2), BogusBiter (Chapter 3), and SessionMagnifier (Chapter 5), I will propose other new approaches to protect users' username and password credentials against various iden-

tity theft malware. In addition to enhancing Web browsing security, I will continue to do research on collaborative browsing and mobile browsing, and I also plan to study cloud computing. Web browsers have increasingly become the dominant platform for both current and future end-user applications, and I believe that there are many interesting Web browsing research problems worth investigating in the future.

Bibliography

- [1] BEN ADIDA. BeamAuth: Two-factor web authentication with a bookmark. In *Proceedings of the CCS*, pages 48–57, 2007.
- [2] BEN ADIDA. Sessionlock: securing web sessions against eavesdropping. In *Proceedings of the WWW*, 2008.
- [3] L. AHN, M. BLUM, N. HOPPER, AND J. LANGFORD. CAPTCHA: Using hard AI problems for security. In *Proceedings of the Eurocrypt*, pages 294–311, 2003.
- [4] RAIHAN AL-EKRAM, ARCHANA ADMA, AND OLGA BAYSAL. diffx: an algorithm to detect changes in multi-version xml documents. In *Proceedings of the CASCON*, pages 1–11, 2005.
- [5] DIRK BALFANZ AND EDWARD W. FELTEN. Hand-held computers can be better smart cards. In *Proceedings of the USENIX Security Symposium*, 1999.
- [6] THOMAS BALL AND JAMES R. LARUS. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst.*, 16(4):1319–1360, 1994.
- [7] ADAM BARTH, COLLIN JACKSON, AND JOHN C. MITCHELL. Robust defenses for cross-site request forgery. In *Proceedings of the CCS*, 2008.
- [8] IRA D. BAXTER, ANDREW YAHIN, LEONARDO MOURA, MARCELO SANT’ANNA, AND LORRAINE BIER. Clone detection using abstract syntax trees. In *Proceedings of the ICSM*, 1998.
- [9] L. BENT, M. RABINOVICH, G. M. VOELKER, AND Z. XIAO. Characterization of a large web site population with implications for content delivery. In *Proceedings of the WWW*, pages 522–533, 2004.
- [10] PHILIP BILLE. A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, 337(1-3):217–239, 2005.
- [11] DOMINIK BIRK, MAXIMILLIAN DORNSEIF, SEBASTIAN GAJEK, AND FELIX GRÖBERT. Phishing phishers - tracing identity thieves and money launderer. Technical Report, Horst-Görtz Institute of Ruhr-University of Bochum, 2006.
- [12] ANDREW BORTZ, DAN BONEH, AND PALASH NANDY. Exposing private information by timing web applications. In *Proceedings of the WWW*, pages 621–628, 2007.

- [13] STEFANO CERI, PIERO FRATERNALI, ALDO BONGIO, MARCO BRAMBILLA, SARA COMAI, AND MARISTELLA MATERA. *Designing Data-Intensive Web Applications*. Morgan Kaufmann, ISBN 1-55860-843-5, 2002.
- [14] SCOTT CHAPMAN AND GURPREET DHILLON. Privacy and the internet: the case of doubleclick, inc, 2002.
- [15] SHUO CHEN, JOSE MESEGUER, RALF SASSE, HELEN J. WANG, AND YI-MIN WANG. A systematic approach to uncover security flaws in gui logic. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy (S&P)*, pages 71–85, 2007.
- [16] SONIA CHIASSON, P. C. VAN OORSCHOT, AND ROBERT BIDDLE. A usability study and critique of two password managers. In *Proceedings of the USENIX Security Symposium*, pages 1–16, 2006.
- [17] NEIL CHOU, ROBERT LEDESMA, YUKA TERAGUCHI, AND JOHN C. MITCHELL. Client-side defense against web-based identity theft. In *Proceedings of the NDSS*, 2004.
- [18] DWAIN E. CLARKE, BLAISE GASSEND, THOMAS KOTWAL, MATT BURNSIDE, MARTEN VAN DIJK, SRINIVAS DEVADAS, AND RONALD L. RIVEST. The untrusted computer problem and camera-based authentication. In *Proceedings of the Pervasive Computing*, 2002.
- [19] RACHNA DHAMIJA AND J.D.TYGAR. The battle against phishing: Dynamic security skins. In *Proceedings of the SOUPS*, pages 77–88, 2005.
- [20] RACHNA DHAMIJA, J.D.TYGAR, AND MARTI HEARST. Why phishing works. In *Proceedings of the CHI*, pages 581–590, 2006.
- [21] JULIE S. DOWNS, MANDY B. HOLBROOK, AND LORRIE FAITH CRANOR. Decision strategies and susceptibility to phishing. In *Proceedings of the SOUPS*, pages 79–90, 2006.
- [22] WOJONG SUH (EDITOR). *Web Engineering: Principles And Techniques*. IGI Publishing, ISBN 1-591-40433-9, 2005.
- [23] SERGE EGELMAN, LORRIE FAITH CRANOR, AND JASON HONG. You’ve been warned: An empirical study of the effectiveness of web browser phishing warnings. In *Proceedings of the CHI*, pages 1065–1074, 2008.
- [24] LAURA FALK, ATUL PRAKASH, AND KEVIN BORDERS. Analyzing websites for user-visible security design flaws. In *Proceedings of SOUPS*, pages 117–126, 2008.
- [25] EDWARD W. FELTEN, DIRK BALFANZ, DREW DEAN, AND DAN S. WALLACH. Web Spoofing: An Internet Con Game. In *Proceedings of the 20th National Information Systems Security Conference*, 1997.
- [26] IAN FETTE, NORMAN SADEH, AND ANTHONY TOMASIC. Learning to detect phishing emails. In *Proceedings of the WWW*, pages 649–656, 2007.

- [27] R. FIELDING, J. GETTYS, J. MOGUL, H. FRYSTYK, L. MASINTER, P. LEACH, AND T. BERNERS-LEE. Hypertext Transfer Protocol – HTTP/1.1, RFC 2616, 1999.
- [28] DAVID FLANAGAN. *JavaScript: The Definitive Guide*. O'Reilly Media, ISBN 0-596-10199-6, 2006.
- [29] DINEI FLORÊNCIO AND CORMAC HERLEY. Klassp: Entering passwords on a spyware infected machine using a shared-secret proxy. In *Proceedings of the ACSAC*, 2006.
- [30] DINEI FLORÊNCIO AND CORMAC HERLEY. Password rescue: A new approach to phishing prevention. In *Proceedings of the HOTSEC*, 2006.
- [31] DINEI FLORÊNCIO AND CORMAC HERLEY. A large-scale study of web password habits. In *Proceedings of the WWW*, pages 657–666, 2007.
- [32] DINEI FLORÊNCIO, CORMAC HERLEY, AND BARIS COSKUN. Do strong web passwords accomplish anything? In *Proceedings of the HOTSEC*, 2007.
- [33] SETH FOGIE, JEREMIAH GROSSMAN, ROBERT HANSEN, ANTON RAGER, AND PETKO D. PETKOV. *XSS Exploits: Cross Site Scripting Attacks and Defense*. Synpress, ISBN 1-597-49154-3, 2007.
- [34] KEVIN FU, EMIL SIT, KENDRA SMITH, AND NICK FEAMSTER. Do's and donts of client authentication on the web. In *Proceedings of the USENIX Security Symposium*, 2001.
- [35] SUJATA GARERA, NIELS PROVOS, MONICA CHEW, AND AVIEL D. RUBIN. A framework for detection and measurement of phishing attacks. In *Proceedings of the WORM*, 2007.
- [36] SCOTT GARRISS, RÁMON CÁCERES, STEFAN BERGER, REINER SAILER, LEENDERT VAN DOORN, AND XIAOLAN ZHANG. Trustworthy and personalized computing on public kiosks. In *Proceedings of the MobiSys*, 2008.
- [37] JEREMY GOECKS AND ELIZABETH D. MYNATT. Social approaches to end-user privacy management. In *Security and Usability: Designing Secure Systems That People Can Use*, 2005.
- [38] SUHIT GUPTA, GAIL KAISER, DAVID NEISTADT, AND PETER GRIMM. Dom-based content extraction of html documents. In *Proceedings of the WWW*, pages 207–214, 2003.
- [39] VICKI HA, KORI INKPEN, FARAH AL SHAAR, AND LINA HDEIB. An examination of user perception and misconception of internet cookies. In *CHI'06 extended abstracts on Human factors in computing systems*, pages 833–838, 2006.
- [40] J. ALEX HALDERMAN, BRENT WATERS, AND EDWARD W. FELTEN. A convenient method for securely managing passwords. In *Proceedings of the WWW*, pages 471–479, 2005.

- [41] YAO-WEN HUANG, FANG YU, CHRISTIAN HANG, CHUNG-HUNG TSAI, DER-TSAI LEE, AND SY-YEN KUO. Securing web application code by static analysis and runtime protection. In *Proceedings of the WWW*, pages 40–52, 2004.
- [42] COLLIN JACKSON AND ADAM BARTH. Forcehttps: protecting high-security web sites from network attacks. In *Proceedings of the WWW*, 2008.
- [43] COLLIN JACKSON, DAN BONEH, AND JOHN MITCHELL. Transaction generators: root kits for web. In *Proceedings of the HOTSEC*, pages 1–4, 2007.
- [44] COLLIN JACKSON, ANDREW BORTZ, DAN BONEH, AND JOHN C. MITCHELL. Protecting browser state from web privacy attacks. In *Proceedings of the WWW*, pages 737–744, 2006.
- [45] TOM N. JAGATIC, NATHANIEL A. JOHNSON, MARKUS JAKOBSSON, AND FILIPPO MENCZER. Social phishing. *Commun. ACM*, 50(10):94–100, 2007.
- [46] MARKUS JAKOBSSON AND STEVEN MYERS. *Phishing and Countermeasures: Understanding the Increasing Problem of Electronic Identity Theft*. Wiley-Interscience, ISBN 0-471-78245-9, 2006.
- [47] MARKUS JAKOBSSON AND SID STAMM. Invasive browser sniffing and countermeasures. In *Proceedings of the WWW*, pages 523–532, 2006.
- [48] MARKUS JAKOBSSON AND ADAM YOUNG. Distributed phishing attacks. In *Proceedings of the workshop on Resilient Financial Information Systems*, 2005.
- [49] RAVI CHANDRA JAMMALAMADAKA, TIMOTHY W. VAN DER HORST, SHARAD MEHROTRA, KENT E. SEAMONS, AND NALINI VENKASUBRAMANIAN. Delegate: A proxy based architecture for secure website access from an untrusted machine. In *Proceedings of the ACSAC*, 2006.
- [50] TAO JIANG, LUSHENG WANG, AND KAIZHONG ZHANG. Alignment of trees - an alternative to tree edit. *Theor. Comput. Sci.*, 143(1):137–148, 1995.
- [51] TREVOR JIM, NIKHIL SWAMY, AND MICHAEL HICKS. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the WWW*, pages 601–610, 2007.
- [52] SACHINDRA JOSHI, NEERAJ AGRAWAL, RAGHU KRISHNAPURAM, AND SUMIT NEGI. A bag of paths model for measuring structural similarity in web documents. In *Proceedings of the KDD*, pages 577–582, 2003.
- [53] STEFAN KALS, ENGIN KIRDA, CHRISTOPHER KRUEGEL, AND NENAD JOVANOVIĆ. Secubat: a web vulnerability scanner. In *Proceedings of the WWW*, pages 247–256, 2006.
- [54] SRIKANTH KANDULA, DINA KATABI, MATTHIAS JACOB, AND ARTHUR W. BERGER. Botz-4-Sale: Surviving Organized DDoS Attacks That Mimic Flash Crowds. In *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*, pages 287–300, 2005.

- [55] GERTI KAPPEL, BIRGIT PROLL, SIEGRIED REICH, AND WERNER RETSCHITZEGGER (EDS.). *Web Engineering: The Discipline of Systematic Development of Web Applications*. John Wiley & Sons, ISBN 0-470-01554-3, 2006.
- [56] ENGIN KIRDA AND CHRISTOPHER KRUEGEL. Protecting users against phishing attacks with AntiPhish. In *Proceedings of the COMPSAC*, pages 517–524, 2005.
- [57] ENGIN KIRDA, CHRISTOPHER KRUEGEL, GREG BANKS, GIOVANNI VIGNA, AND RICHARD A. KEMMERER. Behavior-based spyware detection. In *Proceedings of the USENIX Security Symposium*, pages 273–288, 2006.
- [58] ENGIN KIRDA, CHRISTOPHER KRUEGEL, GIOVANNI VIGNA, AND NENAD JOVANOVIC. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *Proceedings of the 2006 ACM symposium on Applied computing (SAC)*, pages 330–337, 2006.
- [59] DANIEL V. KLEIN. Foiling the cracker – A survey of, and improvements to, password security. In *Proceedings of the 2nd USENIX Workshop on Security*, pages 5–14, 1990.
- [60] BALACHANDER KRISHNAMURTHY AND CRAIG E. WILLS. Cat and mouse: content delivery tradeoffs in web access. In *Proceedings of the WWW*, pages 337–346, 2006.
- [61] D. KRISTOL AND L. MONTULLI. HTTP State Management Mechanism, RFC 2109, 1997.
- [62] D. KRISTOL AND L. MONTULLI. HTTP State Management Mechanism, RFC 2965, 2000.
- [63] DAVID M. KRISTOL. Http cookies: Standards, privacy, and politics. *ACM Trans. Inter. Tech.*, 1(2):151–198, 2001.
- [64] PONNURANGAM KUMARAGURU, YONG RHEE, ALESSANDRO ACQUISTI, LORRIE FAITH CRANOR, JASON HONG, AND ELIZABETH NUNG. Protecting people from phishing: The design and evaluation of an embedded training email system. In *Proceedings of the CHI*, pages 905–914, 2007.
- [65] V. T. LAM, S. ANTONATOS, P. AKRITIDIS, AND K. G. ANAGNOSTAKIS. Puppetnets: misusing web browsers as a distributed attack infrastructure. In *Proceedings of the CCS*, pages 221–234, 2006.
- [66] BENJAMIN LIVSHITS AND WEIDONG CUI. Spectator: detection and containment of javascript worms. In *Proceedings of the USENIX Annual Technical Conference*, 2008.
- [67] MIKE TER LOUW, JIN SOON LIM, AND V.N. VENKATAKRISHNAN. Extensible Web Browser Security. In *Proceedings of the DIMVA*, 2007.
- [68] CHRISTIAN LUDL, SEAN MCALLISTER, ENGIN KIRDA, AND CHRISTOPHER KRUEGEL. On the effectiveness of techniques to detect phishing sites. In *Proceedings of the DIMVA*, 2007.

- [69] BRUCE A. MAH. An empirical model of http network traffic. In *Proceedings of the INFOCOM*, pages 592–600, 1997.
- [70] MOHAMMAD MANNAN AND PAUL C. VAN OORSCHOT. Using a personal device to strengthen password authentication from an untrusted computer. In *Proceedings of the Financial Cryptography*, 2007.
- [71] N. BORIS MARGOLIN, MATTHEW WRIGHT, AND BRIAN NEIL LEVINE. Guardian: A framework for privacy control in untrusted environments. Technical Report, University of Massachusetts, Amherst, 2004.
- [72] JONATHAN M. MCCUNE, ADRIAN PERRIG, AND MICHAEL K. REITER. Bump in the ether: a framework for securing sensitive user input. In *Proceedings of the USENIX Annual Technical Conference*, 2006.
- [73] EMILIA MENDES AND NILE MOSLEY (EDS.). *Web Engineering*. Springer, ISBN 3-540-28196-7, 2005.
- [74] LYNETTE I. MILLETT, BATYA FRIEDMAN, AND EDWARD FELTEN. Cookies and web browser design: toward realizing informed consent online. In *Proceedings of the CHI*, pages 46–52, 2001.
- [75] FABIAN MONROSE, MICHAEL K. REITER, AND SUSANNE WETZEL. Password hardening based on keystroke dynamics. In *Proceedings of the CCS*, pages 73–82, 1999.
- [76] K. MOORE AND N. FREED. Use of HTTP State Management, RFC 2964, 2000.
- [77] TYLER MOORE AND RICHARD CLAYTON. Examining the impact of website take-down on phishing. In *Proceedings of the APWG eCrime Researchers Summit*, 2007.
- [78] ROBERT MORRIS AND KEN THOMPSON. Password security: a case history. *Commun. ACM*, 22(11):594–597, 1979.
- [79] ALEX MOSHCHUK, TANYA BRAGIN, STEVEN D. GRIBBLE, AND HENRY M. LEVY. A crawler-based study of spyware in the web. In *Proceedings of the NDSS*, 2006.
- [80] ALEXANDER MOSHCHUK, TANYA BRAGIN, DAMIEN DEVILLE, STEVEN D. GRIBBLE, AND HENRY M. LEVY. Spyproxy: Execution-based detection of malicious web content. In *Proceedings of the USENIX Security Symposium*, pages 27–42, 2007.
- [81] SAN MURUGESAN AND YOGESH DESHPANDE (EDS.). *Web Engineering : Managing Diversity and Complexity of Web Application Development*. Springer, ISBN 3-540-42130-0, 2001.
- [82] TERRI ODA, GLENN WURSTER, PAUL VAN OORSCHOT, AND ANIL SOMAYAJI. Soma: Mutual approval for included content in web pages. In *Proceedings of the CCS*, 2008.
- [83] ALINA OPREA, DIRK BALFANZ, GLENN DURFEE, AND D. K. SMETTERS. Securing a remote terminal application with a mobile trusted device. In *Proceedings of the ACSAC*, 2004.

- [84] BRYAN PARNO, CYNTHIA KUO, AND ADRIAN PERRIG. Phoolproof phishing prevention. In *Proceedings of the Financial Cryptography*, pages 1–19, 2006.
- [85] BENNY PINKAS AND TOMAS SANDER. Securing passwords against dictionary attacks. In *Proceedings of the CCS*, pages 161–170, 2002.
- [86] THOMAS A. POWELL, DAVID L. JONES, AND DOMINIQUE C. CUTTS. *Web Site Engineering: Beyond Web Page Design*. Prentice Hall, ISBN: 0-13650-920-7, 1998.
- [87] NIELS PROVOS, PANAYIOTIS MAVROMMATIS, MOHEEB ABU RAJAB, AND FABIAN MONROSE. All your iframes point to us. In *Proceedings of the USENIX Security Symposium*, 2008.
- [88] CHARLES REIS, JOHN DUNAGAN, HELEN J. WANG, OPHER DUBROVSKY, AND SAHER ESMEIR. Browsershield: vulnerability-driven filtering of dynamic html. In *Proceedings of the USENIX OSDI*, pages 61–74, 2006.
- [89] D. C. REIS, P. B. GOLGHER, A. S. SILVA, AND A. F. LAENDER. Automatic web news extraction using tree edit distance. In *Proceedings of the WWW*, pages 502–511, 2004.
- [90] TRISTAN RICHARDSON, QUENTIN STAFFORD-FRASER, KENNETH R. WOOD, AND ANDY HOPPER. Virtual network computing. *IEEE Internet Computing*, 2(1):33–38, 1998.
- [91] BLAKE ROSS, COLLIN JACKSON, NICK MIYAKE, DAN BONEH, AND JOHN C MITCHELL. Stronger password authentication using browser extensions. In *Proceedings of the USENIX Security Symposium*, pages 17–32, 2005.
- [92] STEVEN J. ROSS, JASON L. HILL, MICHAEL Y. CHEN, ANTHONY D. JOSEPH, DAVID E. CULLER, AND ERIC A. BREWER. A composable framework for secure multi-modal access to internet services from post-pc devices. *Mob. Netw. Appl.*, 7(5):389–406, 2002.
- [93] GUSTAVO ROSSI, OSCAR PASTOR, DANIEL SCHWABE, AND LUIS OLSINA (EDS.). *Web Engineering: Modelling and Implementing Web Applications*. Springer, ISBN: 1-84628-922-X, 2007.
- [94] STEFAN SAROIU, STEVEN D. GRIBBLE, AND HENRY M. LEVY. Measurement and analysis of spywave in a university environment. In *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*, pages 141–153, 2004.
- [95] STUART E. SCHECHTER, RACHNA DHAMIJA, ANDY OZMENT, AND IAN FISCHER. The emperor’s new security indicators: An evaluation of website authentication and the effect of role playing on usability studies. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 51–65, 2007.
- [96] STANLEY M. SELKOW. The tree-to-tree editing problem. *Inf. Process. Lett.*, 6(6):184–186, 1977.

- [97] UMESH SHANKAR AND CHRIS KARLOF. Doppelganger: Better browser privacy without the bother. In *Proceedings of the ACM CCS*, 2006.
- [98] RICHARD SHARP, ANIL MADHAVAPEDDY, ROY WANT, AND TREVOR PERING. Enhancing web browsing security on public terminals using mobile composition. In *Proceeding of the MobiSys*, 2008.
- [99] RICHARD SHARP, JAMES SCOTT, AND ALASTAIR R. BERESFORD. Secure mobile computing via public terminals. In *Proceedings of the Pervasive Computing*, 2006.
- [100] STEVE SHENG, BRYANT MAGNIEN, PONNURANGAM KUMARAGURU, ALESSANDRO ACQUISTI, LORRIE FAITH CRANOR, JASON HONG, AND ELIZABETH NUNGE. Anti-Phishing Phil: the design and evaluation of a game that teaches people not to fall for phish. In *Proceedings of the SOUPS*, pages 88–99, 2007.
- [101] KUO-CHUNG TAI. The tree-to-tree correction problem. *J. ACM*, 26(3):422–433, 1979.
- [102] E. TANAKA AND K. TANAKA. The tree-to-tree editing problem. *International journal Pattern Recognition And Artificial Intelligency*, 2(2):221–240, 1988.
- [103] ANDREW F. TAPPENDEN AND JAMES MILLER. Cookies: A deployment study and the testing implications. *ACM Trans. Web*, 3(3):1–49, 2009.
- [104] ANDREA TORSELLO AND DZENA HIDOVIC-ROWE. Polynomial-time metrics for attributed trees. *IEEE Trans. Pattern Anal. Mach. Intell.*, 27(7):1087–1099, 2005.
- [105] GABRIEL VALIENTE. An efficient bottom-up distance between trees. In *Proceedings of the SPIRE*, pages 212–219, 2001.
- [106] PHILIPP VOGT, FLORIAN NENTWICH, NENAD JOVANOVIĆ, ENGIN KIRDA, CHRISTOPHER KRUEGEL, AND GIOVANNI VIGNA. Cross site scripting prevention with dynamic data tainting and static analysis. In *Proceedings of the NDSS*, 2007.
- [107] YI-MIN WANG, DOUG BECK, XUXIAN JIANG, ROUSSI ROUSSEV, CHAD VERBOWSKI, SHUO CHEN, AND SAMUEL T. KING. Automated web patrol with Strider HoneyMonkeys: Finding web sites that exploit browser vulnerabilities. In *Proceedings of the NDSS*, 2006.
- [108] ROY WANT, TREVOR PERING, GUNNER DANNEELS, MUTHU KUMAR, MURALI SUNDAR, AND JOHN LIGHT. The personal server: Changing the way we think about ubiquitous computing. In *Proceedings of the Ubicomp*, 2002.
- [109] GARY WASSERMANN AND ZHENDONG SU. Static detection of cross-site scripting vulnerabilities. In *Proceedings of the ICSE*, pages 171–180, 2008.
- [110] C. A. WELTY. Augmenting abstract syntax trees for program understanding. In *Proceedings of the ASE*, 1997.
- [111] TARA WHALEN AND KORI M. INKPEN. Gathering evidence: use of visual security cues in web browsers. In *Proceedings of the conference on Graphics interface*, pages 137–144, 2005.

- [112] MIN WU. *Fighting Phishing at the User Interface*. PhD thesis, MIT, 2006.
- [113] MIN WU, SIMSON GARFINKEL, AND ROB MILLER. Secure web authentication with mobile phones. In *Proceedings of the DIMACS Workshop on Usable Privacy and Security Software*, 2004.
- [114] MIN WU, ROBERT C. MILLER, AND SIMSON L. GARFINKEL. Do security toolbars actually prevent phishing attacks? In *Proceedings of the CHI*, pages 601–610, 2006.
- [115] MIN WU, ROBERT C. MILLER, AND GREG LITTLE. Web Wallet: preventing phishing attacks by revealing user intentions. In *Proceedings of the SOUPS*, pages 102–113, 2006.
- [116] THOMAS WU. The secure remote password protocol. In *Proceedings of the NDSS*, 1998.
- [117] WUU YANG. Identifying syntactic differences between two programs. *Softw. Pract. Exper.*, 21(7):739–755, 1991.
- [118] ZISHUANG (EILEEN) YE AND SEAN SMITH. Trusted paths for browsers. In *Proceedings of the USENIX Security Symposium*, pages 263–279, 2002.
- [119] KA-PING YEE AND KRAGEN SITAKER. Passpet: convenient password management and phishing protection. In *Proceedings of the SOUPS*, pages 32–43, 2006.
- [120] DACHUAN YU, AJAY CHANDER, NAYEEM ISLAM, AND IGOR SERIKOV. Javascript instrumentation for browser security. In *Proceedings of the POPL*, pages 237–249, 2007.
- [121] CHUAN YUE, ZI CHU, AND HAINING WANG. RCB: A Simple and Practical Framework for Real-time Collaborative Browsing. In *Proceedings of the USENIX Annual Technical Conference*, pages 369–382, 2009.
- [122] CHUAN YUE AND HAINING WANG. Anti-Phishing in Offense and Defense. In *Proceedings of the ACSAC*, pages 345–354, 2008.
- [123] CHUAN YUE AND HAINING WANG. Characterizing Insecure JavaScript Practices on the Web. In *Proceedings of the WWW*, pages 961–970, 2009.
- [124] CHUAN YUE AND HAINING WANG. SessionMagnifier: A Simple Approach to Secure and Convenient Kiosk Browsing. In *Proceedings of the Ubicomp*, pages 125–134, 2009.
- [125] CHUAN YUE AND HAINING WANG. BogusBiter: A Transparent Protection Against Phishing Attacks. *ACM Trans. Internet Technol.*, 10(2):1–31, 2010.
- [126] CHUAN YUE, MENGJUN XIE, AND HAINING WANG. Automatic Cookie Usage Setting with CookiePicker. In *Proceedings of the Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 460–470, 2007.
- [127] CHUAN YUE, MENGJUN XIE, AND HAINING WANG. An Automatic HTTP Cookie Management System. *Journal of Computer Networks (COMNET)*, Elsevier, 2010.

- [128] YANHONG ZHAI AND BING LIU. Web data extraction based on partial tree alignment. In *Proceedings of the WWW*, pages 76–85, 2005.
- [129] YANHONG ZHAI AND BING LIU. Structured data extraction from the web based on partial tree alignment. *IEEE Transactions on Knowledge and Data Engineering*, 18(12):1614–1628, 2006.
- [130] K. ZHANG AND D. SHASHA. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18(6):1245–1262, December 1989.
- [131] YUE ZHANG, SERGE EGELMAN, LORRIE FAITH CRANOR, AND JASON HONG. Phinding phish: Evaluating anti-phishing tools. In *Proceedings of the NDSS*, 2007.
- [132] YUE ZHANG, JASON HONG, AND LORRIE CRANOR. CANTINA: A content-based approach to detecting phishing web sites. In *Proceedings of the WWW*, pages 639–648, 2007.
- [133] 24 ways: Don't be eval(). <http://24ways.org/2005/dont-be-eval>.
- [134] 5 safety tips for using a public computer. <http://www.microsoft.com/protect/yourself/mobile/publicpc.mspx>.
- [135] Ajax(programming). [http://en.wikipedia.org/wiki/Ajax_\(programming\)](http://en.wikipedia.org/wiki/Ajax_(programming)).
- [136] Alexa Top Sites. <http://www.alexa.com/browse?CategoryID=1>.
- [137] Anti-Phishing Working Group (APWG). <http://www.antiphishing.org/>.
- [138] APWG: Phishing Scams by Targeted Company. <http://www.millersmiles.co.uk/scams.php>.
- [139] Cookie Crusher. <http://www.pcworld.com/downloads>.
- [140] Cookie Culler. <http://cookieculler.mozdev.org>.
- [141] Cookie pal. <http://www.kburra.com/cpal.html>.
- [142] Cross-site scripting. http://en.wikipedia.org/wiki/Cross-site_scripting.
- [143] Data Flow Inside Gecko. <http://developer.mozilla.org/en/docs>.
- [144] eBanking Security. http://www.ebankingsecurity.com/ebanking_bad_for_your_bank_balance.pdf.
- [145] eval-MDC. http://developer.mozilla.org/en/Core_JavaScript_1.5_Reference/Global_Functions/eval.
- [146] Firefox 2 Phishing Protection Effectiveness Testing. <http://www.mozilla.org/security/phishing-test.html>.
- [147] Firefox Browser Extensions. <http://developer.mozilla.org>.

- [148] Firefox Phishing Protection. <http://www.mozilla.com/en-US/firefox/phishing-protection/>.
- [149] Gartner, Inc. Survey. <http://www.gartner.com/it/page.jsp?id=498245>.
- [150] GNU Wget - GNU Project - Free Software Foundation (FSF). <http://www.gnu.org/software/wget/>.
- [151] Gone Phishing: Evaluating Anti-Phishing Tools for Windows. <http://www.3sharp.com/projects/antiphishing/gone-phishing.pdf>.
- [152] Hacker demos how to defeat Citibanks virtual keyboard. <http://blogs.zdnet.com/security/?p=195>.
- [153] HMAC. <http://en.wikipedia.org/wiki/HMAC>.
- [154] HTML 4.01 Specification. <http://www.w3.org/TR/html4/>.
- [155] HTTP Cookie. http://en.wikipedia.org/wiki/HTTP_cookie.
- [156] IBM set to use spam to attack spammer. http://money.cnn.com/2005/03/22/technology/ibm_spam/index.htm.
- [157] IE Browser Extensions. [http://msdn.microsoft.com/en-us/library/aa753587\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/aa753587(VS.85).aspx).
- [158] Inaccessibility of CAPTCHA. <http://www.w3.org/TR/turingtest/>.
- [159] Internet Security Threat Report, Security research and analysis — Symantec. <http://www.symantec.com/business/theme.jsp?themeid=threatreport>.
- [160] JavaScript. <http://en.wikipedia.org/wiki/JavaScript>.
- [161] JSAPI reference-MDC. http://developer.mozilla.org/en/JSAPI_Reference.
- [162] JSON in JavaScript. <http://www.json.org/js.html>.
- [163] JSPrincipals-MDC. <http://developer.mozilla.org/en/JSPrincipals>.
- [164] jssha256. <http://point-at-infinity.org/jssha256/>.
- [165] Know Your Enemy: Phishing. <http://www.honeynet.org/papers/phishing/>.
- [166] Likert scale. http://en.wikipedia.org/wiki/Likert_scale.
- [167] MarkMonitor: Internet Fraud Prevention and Brand Protection. <http://www.markmonitor.com/>.
- [168] Microsoft ActiveSync. <http://www.microsoft.com/windowsmobile/en-us/help/synchronize/device-synch.mspx>.
- [169] Microsoft Phishing Filter. <http://www.microsoft.com/protect/products/yourself/>.

- [170] Mitigating Cross-site Scripting with HTTP-only Cookies. <http://msdn2.microsoft.com/en-us/library/ms533046.aspx>.
- [171] Mobile/Fennec. <https://wiki.mozilla.org/Fennec>.
- [172] Mozilla XUL - MDC. <http://xulplanet.com/references/xpcomref>.
- [173] MSDN: innerHTML property. [http://msdn.microsoft.com/en-us/library/ms533897\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms533897(VS.85).aspx).
- [174] Online identity theft is the greatest fear for internet users. http://www.securitypark.co.uk/security_article264484.html.
- [175] Permit Cookies. <https://addons.mozilla.org/firefox/44>.
- [176] PhishTank. <http://www.phishtank.com/>.
- [177] Platform for privacy preferences (P3P) project. <http://www.w3.org/P3P/>.
- [178] Privoxy - Home Page. <http://www.privoxy.org/>.
- [179] RSA, The Security Division of EMC. <http://www.rsa.com/>.
- [180] Same origin policy. http://en.wikipedia.org/wiki/Same_origin_policy.
- [181] SANS Top-20 2007 Security Risks (2007 Annual Update). <http://www.sans.org/top20/2007/>.
- [182] Semantic Attacks: The Third Wave of Network Attacks. <http://www.schneier.com/crypto-gram-0010.html#1>.
- [183] SpiderMonkey (JavaScript-C) Engine. <http://www.mozilla.org/js/spidermonkey/>.
- [184] Symantec Internet Security Threat Report Volume XIII: April, 2008. <http://www.symantec.com/business/theme.jsp?themeid=threatreport>.
- [185] tcpmon: An open-source utility to Monitor A TCP Connection. <https://tcpmon.dev.java.net/>.
- [186] Tim Berners-Lee: WorldWideWeb, the first Web client. <http://www.w3.org/People/Berners-Lee/WorldWideWeb.html>.
- [187] Trusted Computing Group. <https://www.trustedcomputinggroup.org>.
- [188] Understanding and countering the phishing threat, the financial services technology consortium (fstc) project white paper. http://fstc.org/projects/counter_phishing_phase_1/.
- [189] Unobtrusive Javascript. <http://www.onlinetools.org/articles/unobtrusivejavascript/>.

- [190] Vulnerability Type Distribution in CVE. <http://www.attrition.org/pipermail/vim/2006-September/001032.html>.
- [191] W3C Document Object Model. <http://www.w3.org/DOM>.
- [192] XMLHttpRequest. <http://www.w3.org/TR/XMLHttpRequest/>.
- [193] CERT Advisory CA-2000-02 Malicious HTML tags embedded in client web requests, 2000. <http://www.cert.org/advisories/CA-2000-02.html>.
- [194] Accurate web site visitor measurement crippled by cookie blocking and deletion, jupiterresearch finds, 2005. <http://www.jupitermedia.com/corporate/releases/05.03.14-newjupresearch.html>.
- [195] Google plugs cookie-theft data leak, 2005. <http://www.eweek.com/article2/0,1895,1751689,00.asp>.
- [196] Flaws in IE7 and Firefox raise alarm, 2007. <http://news.zdnet.co.uk/security,February19th,2007>.
- [197] Google slams the door on XSS flaw 'Stop cookie thief!', 2007. <http://software.silicon.com/security/,January17th,2007>.

VITA

Chuan Yue

Chuan Yue enrolled in the computer science Ph.D. program at The College of William and Mary in Fall 2004. His broad research interests include Web-based systems, computer and information security, distributed and parallel computing, human-computer interaction, and collaborative computing. His Ph.D. research focuses on Web browsing security and collaborative browsing. Previously, he received his B.E. and M.E. degrees in computer science from Xidian University in 1996 and 1999, respectively; he then worked as a Member of Technical Staff at Bell Labs China, Lucent Technologies from 1999 to 2003, mainly on the development of Web-based distributed service management system for Intelligent Network.