MODERN TECHNIQUES FOR THE DETECTION AND PREVENTION OF WEB2.0 ATTACKS

Elias Athanasopoulos

Submitted in partial fulfillment of the requirements

for the degree of Doctor of Philosophy

in Computer Science

in the Graduate Division

of the University of Crete

Heraklion, June 2011

Modern Techniques for the Detection and Prevention of Web2.0 Attacks

A dissertation submitted by
Elias Athanasopoulos
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate Division of the University of Crete

The dissertation of Elias Athanasopoulos is approved:

Heraklion, June 2011

Abstract

In this dissertation we examine web exploitation from a number of different perspectives. First, we introduce *return-to-JavaScript* attacks; a new flavor of Cross-Site Scripting (XSS), which is able to escape script whitelisting. Second, we design xJS, a system that can prevent code injections of JavaScript in web applications. xJS is based on the concept of Instruction Set Randomization (ISR) for isolating legitimate JavaScript from malicious injections. We evaluate xJS and show that the overhead it imposes in the server's and the client's side is negligible, since xJS is based on the fast `XOR` operation. Third, we deliver a more fine-grained randomization framework for web applications, RaJa, which can efficiently cope with language mixing. RaJa can successfully extract and randomize the JavaScript source code of real-world applications, which experience heavy code-mixing (i.e. JavaScript is mixed with a server-side programming language, such as PHP). Forth, we present xHunter, a network-level detector, which is able to locate JavaScript fragments in the body of URLs. With the assistance of xHunter we deliver an extensive analysis of the largest to date web-attack repository, XSSed.com. This particular repository hosts about 12,000 incidents of web exploitation. Our analysis identifies that 7% of all examined web attacks do not use any markup elements, such as `<script>` or `<iframe>`, for exploiting a web application. All these incidents are hard to be captured by tools that are based on static signatures and regular expressions. We proceed and analyze these attacks with xHunter and we deliver a compact signature set, which is composed by no more than a handful of rules. This rule-set is based on signatures expressed as JavaScript syntax trees and can be used to detect a broad family of injections that target web applications.

Finally, we address the problem of data fabrication in data collected by web servers, VoIP providers, on-line stores and ISPs. We present Network Flow Contracts (NFCs), a system, which, in spite of the presence of malicious web servers or untrusted ISPs, enables innocent users to prove that they have not accessed the illegal content in question. NFCs require every network request to be cryptographically signed by the requesting user. We present a prototype implementation as well as a performance evaluation on top of commodity technologies.

The results of this research are the followings. First, Instruction Set Randomization can be efficiently applied in web applications with low performance overhead and large attack coverage. Second, web-attack detection at the network level is also possible, although computationally expensive to be applied in real-time. Third, cryptographically signed network flows can protect users from data fabrication at the ISP level with low cost.

Thesis Advisor: Prof. Evangelos Markatos

Περίληψη

Σε αυτή τη διατριβή εξετάζουμε επιθέσεις στον Παγκόσμιο Ιστό από μια σειρά διαφορετικών οπτικών γωνιών. Πρώτον, εισάγουμε τις επιθέσεις $return-to-JavaScript$, μια νέα μορφή $Cross-Site\ Scripting(XSS)$, η οποία δύναται να ξεπεράσει μέτρα που βασίζονται σε *script whitelisting*. Δεύτερον, σχεδιάζουμε την αρχιτεκτονική ενός πλήρους συστήματος, xJS, το οποίο μπορεί να αποτρέψει επιθέσεις εισαγωγής κώδικα $JavaScript$ σε εφαρμογές Παγκόσμιου Ιστού. Το xJS βασίζεται στην ιδέα της Τυχαιοποίησης Σετ Εντολών (ΤΣΕ), έτσι ώστε να μπορεί να απομονώσει νόμιμη $JavaScript$ από κακόβουλες εισαγωγές. Κατά την αποτίμηση του xJS δείχνουμε ότι η επιβάρυνση στην πλευρά του διακομιστή και στην πλευρά του πελάτη είναι αμελητέα, μιας και το xJS βασίζεται στη γρήγορη εντολή XOR. Τρίτον, κατασκευάζουμε ένα πλαίσιο τυχαιοποίησης για εφαρμογές Παγκόσμιου Ιστού, $RaJa$, το οποίο μπορεί να αντεπεξέλθει σε περιβάλλοντα όπου γλώσσες, όπως π.χ. η PHP και η $JavaScript$, αναμιγνύονται μεταξύ τους. Τέταρτον, παρουσιάζουμε το $xHunter$, έναν ανιχνευτή επιπέδου δικτύου, ο οποίος μπορεί να εντοπίσει κομμάτια $JavaScript$ σε $URLs$. Με τη βοήθεια του $xHunter$ πραγματοποιούμε μια εκτεταμένη ανάλυση της μεγαλύτερης έως σήμερα πηγής επιθέσεων Παγκόσμιου Ιστού, $XSSed.com$, η οποία φιλοξενεί περίπου 12,000 συμβάντα. Η ανάλυσή μας αποδεικνύει ότι στο 7% όλων των επιθέσεων δε χρησιμοποιούνται συστατικά $Markup$, όπως $<script>$ και $<iframe>$. Τέτοιου είδους επιθέσεις είναι πολύ δύσκολο να ανιχνευθούν από εργαλεία βασισμένα σε στατικές υπογραφές. Αναλύουμε όλες αυτές τις επιθέσεις με το $xHunter$ και παρουσιάζουμε ένα σετ μερικών δεκάδων υπογραφών. Το σετ βασίζεται σε υπογραφές, οι οποίες εκφράζονται με τη βοήθεια συντακτικών δένδρων $JavaScript$.

Τέλος, αντιμετωπίζουμε το πρόβλημα της αλλοίωσης δεδομένων, τα οποία συλλέγονται από διακομιστές Παγκόσμιου Ιστού, παροχείς υπηρεσιών $VoIP$, ψηφιακών καταστημάτων και Παροχείς Υπηρεσιών Ίντερνετ. Παρουσιάζουμε τα Συμβόλαια Δικτυακών Ροών (ΣΔΡ), ένα σύστημα το οποίο μπορεί να δώσει τη δυνατότητα σε χρήστες να αποδείξουν ότι δεν έχουν αποπειραθεί να έρθουν σε επαφή με παράνομο περιεχόμενο. Τα ΣΔΡ απαιτούν κάθε δικτυακή αίτηση να είναι κρυπτογραφικά υπογεγραμμένη. Παρουσιάζουμε μια πρωτότυπη υλοποίηση, όπως και αποτίμηση της απόδοσής των ΣΔΡ.

Το αποτέλεσμα αυτής της έρευνας είναι ότι η Τυχαιοποίηση Σετ Εντολών μπορεί να εφαρμοσθεί σε εφαρμογές Παγκόσμιου Ιστού με μικρή επιβάρυνση και αντιμετώπιση ενός μεγάλου εύρους επιθέσεων, ενώ ανίχνευση επιθέσεων Παγκόσμιου Ιστού σε επιπέδου Δικτύου είναι δυνατή, αν και υπολογιστικά ακριβή για να εφαρμοστεί σε πραγματικό χρόνο. Κρυπτογραφικά υπογεγραμμένες δικτυακές ροές μπορούν να προστατέψουν χρήστες από την αλλοίωση δεδομένων σε επίπεδο Παροχέα Υπηρεσιών Ίντερνετ με μικρό κόστος.

Επόπτης: Καθηγητής Ευάγγελος Π. Μαρκάτος

The research outlined in this dissertation has been funded by Microsoft Research (Cambridge) and the Microsoft Research Scholarship Program under the title: *On the misuses of real-world large-scale distributed systems.* I was able to complete this dissertation, while I was supervised by both Professor Evangelos Markatos and Dr. Thomas Karagiannis of Microsoft Research. Our in-close collaboration resulted in the following, significant for the largest part of this dissertation's content, publication:

Elias Athanasopoulos, Vasilis Pappas, Antonis Krithinakis, Spyros Ligouras, Evangelos P. Markatos, and Thomas Karagiannis. **xJS: Practical XSS Prevention for Web Application Development.** *In Proceedings of the 1st USENIX Conference on Web Application Development (WebApps).* June 2010, Boston, Massachusetts.

The research outlined in this dissertation has been conducted in the Distributed Computing Systems laboratory, which is hosted in the Institute of Computer Science of Foundation of Research and Technology, Hellas.

Acknowledgments

I would like to thank the people that have significantly influenced and inspired my life during my graduate studies.

First of all, my advisor, Evangelos Markatos, for having strong patience with me. I can recall, and most probably I will never forget of, many memorable debates! I gained invaluable knowledge from him and learned how to scientifically address problems. Most importantly, he convinced me that even "simple tasks" cannot be always solved in *one evening*. Second, my mentor, Thomas Karagiannis of Microsoft Research, who assisted me with his guideline in all the research outlined in this dissertation. Third, the rest of my committee, Sotiris Ioannidis, Mema Roussopoulos, Angelos Bilas, Dimitris Nikolopoulos, and Maria Papadopouli for carrying out professionally the process of my PhD defense.

I consider myself very fortunate for being surrounded by brilliant and beautiful minds. All these years I had the opportunity to take part in very interesting technical and non-technical discussions with Sotiris Ioannidis, Kostas Anagnostakis, Periklis Akritidis, John Ioannidis, Angelos Keromytis, Christos Gkantsidis, and Catherine Chronaki.

Research is fun when you have the opportunity to carry out the hard work while being accompanied by friends. These are Demetres Antoniades, Christos Papachristos, and Michalis Polychronakis, the old school of DCS, Spyros Antonatos, Manos Athanatos, Antonis Papadogiannakis, Manolis Stamatogiannakis, Dimitris Koukis, Manos Moschous, the new blood of DCS, George Vasiliadis, George Kondaxis, Iasonas Polakis, Eleni Gessiou, as well as students I had in-close research with, Vasilis Pappas and Antonis Krithinakis.

All people I met while living in Crete, Autalo and George, Deukalionas and Maria, Chrysa and Manolis, Eleni Milaki, Eva Poniraki, and Charalambos Baharidis.

I wouldn’t have made it so far without the support of my family, my mother, father and brother. A big “thank you” to the newcomer of the family, my little niece Efi, for reminding to all of us that life is truly beautiful.

Eleni, an unexpected gift, who gave me infinite happiness in Crete. She really makes myself impatience for our future...

Στη μητέρα μου.

To my mother.

Contents

1 Introduction **1**

1.1 Problem Statement and Approach 2

1.2 Novelty and Contributions 3

1.3 Dissertation Organization 4

2 Background **6**

2.1 The Web2.0 Era 6

2.2 Security Threats in Web Applications 7

2.2.1 Cross-Site Scripting (XSS) 8

2.2.2 Cross-Site Request Forgery (CSRF) 8

2.2.3 Visual Attacks 9

2.2.4 Other 9

2.3 Instruction Set Randomization 10

3 Return To JavaScript Attacks **11**

3.1 Overview 11

3.2 DOM-based Attacks 14

4 Countering XSS Using Isolation Operators **17**

4.1 Overview 17

4.1.1 Attacks Not Addressed 19

4.2 Architecture 19

4.2.1 Isolation Operators 20

4.2.2 Action Based Policies 21
4.3 Implementation 22
4.4 Evaluation 24
4.4.1 Attack Coverage 24
4.4.2 Server Overhead 27
4.4.3 Client Overhead 29
4.4.4 User Browsing Experience 31
4.5 Discussion 32

5 Applying ISR Directly To JavaScript **34**
5.1 Overview 34
5.2 Architecture 35
5.2.1 Overview 36
5.2.2 Server Side 37
5.2.3 Randomization 40
5.2.4 De-randomization 41
5.2.5 Configuration and Administrator Tools 42
5.3 Case Studies 43
5.3.1 WordPress 43
5.3.2 phpBB, phpMyAdmin and Drupal 46
5.3.3 Summary 47
5.4 Evaluation 47
5.5 Discussion 53

6 xHunter **55**
6.1 Introduction 55
6.2 xHunter Architecture 56
6.2.1 Overview 57
6.2.2 Operation 58
6.3 Challenges 60

6.3.1 Web Application Quirks 60
6.3.2 JavaScript Relaxed Syntax 61
6.3.3 Exploit Isolation 62
6.4 Case Study 63
6.4.1 XSSed.com 64
6.4.2 Benign URLs 66

7 XSS Study **67**

7.1 Introduction 67
7.2 Methodology 69
7.2.1 Overview 69
7.2.2 URL Processing 71
7.3 Experimental Results 74
7.3.1 Exploitation Method 75
7.3.2 Target Characterization 77
7.3.3 Repository Characterization 82
7.4 JavaScript Analysis 83
7.4.1 Results 84
7.4.2 Syntax-Tree Based Signatures 85
7.4.3 Discussion 86

8 Network Flow Contracts **89**

8.1 Overview 89
8.1.1 Proposal 91
8.1.2 Contribution 92
8.2 Threat Model 92
8.2.1 Overview 93
8.2.2 Malicious Web Sites 94
8.2.3 Untrusted ISPs 94
8.3 Architecture 95

8.3.1 Overview 96
8.3.2 Network Flow Contracts 97
8.3.3 Web Traffic 100
8.3.4 Dig Encryption 101
8.4 Evaluation 103
8.4.1 NFCs 104
8.4.2 Server Overhead 109
8.4.3 Dig Encryption 110
8.5 Discussion and Future Work 111
8.5.1 Traceability 111
8.5.2 UDP Traffic 111
8.5.3 RSA Cost 112

9 Related Work **113**
9.1 Web Exploitation and Defenses 113
9.1.1 Data Tainting 114
9.1.2 Sophisticated Web Attacks 114
9.2 Instruction Set Randomization 115
9.3 Anomaly Detection Systems and Regular Expressions 115
9.4 Browser Operating Systems 116
9.5 Network Flow Contracts 116

10 Conclusion **118**
10.1 Using Instruction Set Randomization for XSS Mitigation 118
10.1.1 xJS 119
10.1.2 RaJa 119
10.2 Network Level XSS Detection 120
10.2.1 xHunter 120
10.2.2 JavaScript-based Signatures 120
10.3 Network Flow Contracts 121

A Publications **122**

List of Figures

3.1 A minimal Blog site demonstrating the whitelisting attacks. 13

4.1 Example of a web page that is generated by our framework. On the left, the figure shows the source code as exists in the web server and on the right the same source as it is fetched by the web browser. The JavaScript source has been `XOR`ed and a Base64 encoding has been applied in order to transpose all non-printable characters to the printable ASCII range. 19

4.2 Server side evaluation when the Apache benchmark tool (`ab`) is requesting each web page through a Fast Ethernet link. In the worst case (heavy) the server imposes delay of a factor of five greater, while in the normal case the delay is only a few milliseconds. 28

4.3 Server side evaluation when the Apache benchmark tool (`ab`) is requesting each web page through a DSL link. In the worst case (heavy) the server imposes a fixed delay of a few tens of milliseconds, like in the case of the Fast Ethernet setup (see Figure 4.2). However, this delay does not dominate the overall delivery time. 29

4.4 Cumulative distribution for the delay imposed by all modified function calls in the Firefox and WebKit implementation, respectively. As delay we assume the time needed for the modified function to complete minus the time needed for the unmodified one to complete. Notice that the majority of function calls imposes a delay of a few milliseconds. 30

4.5 Results from the SunSpider test suite. Notice that for each modified browser the results are comparable with the results of its unmodified equivalent. That is, all de-isolated JavaScript executes as expected in both modified and unmodified browser. 31

5.1 PHP mixes with JavaScript. In the majority of the cases, the PHP expression acts as a JavaScript identifier or literal. In this example it acts as a string literal. 35

5.2 A typical RaJa example. 37

5.3 Schematic diagram of the RaJa architecture. 39

5.4 Code mixing of JavaScript with alien languages. In this example PHP is used as an alien language example. In line 3, Rule 1 is applied, while in lines 2, 4 and 6, Rule 2 is applied. 41

5.5 A typical RaJa configuration file. 43

5.6 Categorization of all cases that result in faulty randomization due to interference between JavaScript and PHP. 48

5.7 Suggested workarounds for all cases that result in faulty randomization due to interference between JavaScript and PHP. 49

5.8 Example of randomized source code from WordPress (`wp-login.php`). 50

5.9 Server side evaluation when the Apache benchmark tool (`ab`) is requesting each web page through a Fast Ethernet link. In the worst case (heavy) the server imposes delay of about 100 of milliseconds, while in the normal case the delay is only a few milliseconds. 51

5.10 Server side evaluation when the Apache benchmark tool (`ab`) is requesting each web page through a DSL link. In the worst case (heavy) the server imposes a fixed delay of a few tens of milliseconds, like in the case of the Fast Ethernet setup (see Figure 5.9). However, this delay does not dominate the overall delivery time. 52

5.11 Client-side evaluation for a RaJa-enabled web browser using the SunSpider benchmarks. A RaJa-enabled web browser is slightly slower compared to Firefox. 53

6.1 Parse tree of a JavaScript snippet as produced by xHunter. 58
6.2 Example operation of a hypothetical xHunter run. 59
6.3 Custom encoding used by web applications. In the second example some parts are omitted for better presentation. 61
6.4 JavaScript has a relaxed syntax. Even simple text fragments can produce a syntax tree with high depth. 63
6.5 XSS code can be found partial in URLs. In this example the "″" character (double quotes) must be omitted in order for the rest of the code to produce a valid syntax tree. 64

7.1 The six different categories used for exploit classification by XSSed.com. . . . 70
7.2 Schematic diagram of all steps followed by xHunter for extracting additional information for each web exploit. 71
7.3 The HTTP property (protocol and method) used by each web exploit. 74
7.4 HTML markup used in web exploits. 75
7.5 The rank, as provided by Alexa.com, of web sites that are targets of XSS attacks. 77
7.6 XSS exploits during the last couple of years against all Top-100, based on Alexa.com, web sites. Notice the increase in 2008, which is also reported by Symantec [Sym]. 78
7.7 Days required by a web site to repair the vulnerability after the submission or publication date of an exploit. We depict two sets of web sites, the ones that are ranked in Top-100 by Alexa.com and all the others. Notice, that, in general, the publication date is closer to the date of patching than the submission one. Also, Top-100 web sites seem to be faster in patching. 79
7.8 Web sites listed in XSSed.com as attack targets and have been also experienced defacement incidents as reported by Zone-h.org. At least one defacement incident has been experienced by 14% of the XSSed.com sample. 80
7.9 The Top-10 countries that host the most frequently attacked web sites. 81
7.10 Days required for XSSed.com to verify and publish a submitted web exploit. . 82

7.11 Number of exploits that have been submitted and published each date. 83

7.12 An example of how a JavaScript syntax-tree is extracted from an expression. 84

7.13 Similarity of all JavaScript syntax tree based signatures. The size of the radius represents the Least Common Subsequence (LCS) of a pair of signatures. . . 85

8.1 NFC packet format. The packet is composed by an identifier, ID, which characterizes the connection, a delimiter S and the Base64 encoding of the signature. The ID is formed using the SHA1 key generated by hashing the subscriber name concatenated with the 3-tuple information and the signature is the destination IP encrypted with the user's private key. 95

8.2 The NFC architecture. The ISP maintains for each on-line subscriber two tables: (a) the Connection Table, and (b) the Signature Table. Every new flow is stored in the Connection Table and incoming signatures are stored in the Signature Table. If, after time T, no signature for a new flow is present in the Signature Table the new flow is terminated. 96

8.3 The NFC protocol. For every incoming `TCP-SYN` the ISP waits for the NFC that signs the new flow. In the case that no signature is received after T seconds from the time the original `TCP-SYN` was observed, the ISP terminates the subscriber's connection by sending an explicit `TCP-RST`. 97

8.4 Enabling NFCs for web traffic requires the addition of a new HTTP header, which simply signs with the user's private key the URL resource that the browser is requesting. 98

8.5 NFCs evaluation for an NFC client connected with an NFC server through a Fast Ethernet network. 104

8.6 Overhead of RSA encryption with a private key for different architectures. . . 105

8.7 NFCs evaluation for an NFC client connected with an NFC server through a DSL line. 106

8.8 CDF of the request rate extracted from four web caches obtained by IR-Cache [irc] in the period of one day. Notice, that the average request rate for the heavy loaded cache is a few hundred requests/sec. 107

8.9 Prototype implementation of dig encryption in Ruby. 108

8.10 The time which is taken to *undig* a message for various ranges of salt's space in hours. 109

Chapter 1

Introduction

The World Wide Web (WWW), which we will refer to as the web platform, has evolved into a large-scale system composed by millions of applications and services. In the beginning, there were only static web pages aiming at providing information expressed in text and graphics. Today, the web platform provides alternatives for all native applications that have been deployed in modern operating systems over the last decades. Users can use web applications for communicating with other users via instant messaging, for reading e-mail and news, for editing and viewing video, for managing their photographs and other files, or, even creating spreadsheets, presentations and text documents. Apart from traditional applications, the web provides also new applications, which are hard to develop, without taking advantage of the mass adoption of the web platform. Social networks, such as Facebook and MySpace, blogs and micro-blogs, such as Twitter, and other content providing services that are built on users' collaboration, such as YouTube and Flickr, are considered the *killer applications* of the last few years.

Moreover, the web browser is considered as the de facto medium for accessing and interconnecting all services, from every-day web surfing to financial transactions related to e-banking and to the stock market. Every Internet-connected device is able to run a web browser and, with standard technologies composed by the HTTP protocol, the JavaScript programming language and the XML dialect, these devices deliver a global interface for accessing the web platform. It is highly unlikely that the trend towards developing and using new web applications is going to change in the near future. On the contrary, the evolution of the web platform is so rapid, that has driven large IT companies, such as Google, to create and offer laptops [cr4] equipped with an Operating System [chr] entirely developed over the the web platform. A system that lives *in the cloud*.

1.1 Problem Statement and Approach

Unfortunately, the great evolution of the web platform is also connected with the parallel explosion of security threats that occur in web applications. These threats affect directly all users that take advantage of the web platform. Especially large web applications, such as social network utilities, which promote similar content to a large user base, can affect a significant amount of users in the case they become vulnerable. We are among the first to highlight this issue [AMA+08, MAA+10].

Moreover, code-injection attacks through Cross-Site Scripting (XSS) in the web browser have observed a significant increase over the previous years. According to a September-2009 report published by the SANS Institute [SAN], *attacks against web applications constitute more than 60% of the total attack attempts observed on the Internet. Web application vulnerabilities such as SQL injection and Cross-Site Scripting flaws in open-source as well as custom-built applications account for more than 80% of the vulnerabilities being discovered.* XSS threats are not only targeted towards relatively simple, small-business web sites, but also towards infrastructures that are managed and operated by leading IT vendors [mca]. Moreover, recently widely adopted technologies, such as AJAX [G+05], exacerbate potential XSS vulnerabilities by promoting richer and more complex client-side interfaces. This added complexity in the web browser environment provides additional opportunities for further exploitation of XSS vulnerabilities.

Injecting malicious code using an XSS vulnerability is not the only way to exploit a web application. An attacker can control a web application using a series of techniques, as we show in Chapter 2. However, irrespective of the exploitation method, the great majority of already published web exploits is based on the usage of JavaScript (we quantify this in Chapter 7).

This is not meant to be surprising. Let us think of a web application based on knowledge we have so far from native applications. A native application is composed by the data segment (where data are stored) and the code segment (where code is stored). An attacker who is willing to compromise the application attempts to inject some code, using for example a buffer overflow vulnerability [One96], which will eventually *glue* to the existing code of the program and force the application to behave according to the attacker's needs. In a similar fashion, we can think of a web application as a program, which consists of data (text and graphics expressed in HTML) and JavaScript code, which orchestrates the web application flow in the web browser environment. Thus, a web exploit based on JavaScript gives the attacker rich flexibility in terms of altering the web application's behavior.

In this thesis we propose technologies that can protect web applications from JavaScript injections. We first argue that simple JavaScript whitelisting [1] is not sufficient, by introducing a new XSS flavor; *return-to-JavaScript* attacks. We, then, build frameworks that isolate all legitimate JavaScript of a web application from possible code injections. Our approach is based on a generic principle for countering code injections known as Instruction Set Randomization (ISR). We are the first to apply ISR to web applications. We, furthermore, build a network-level detector, which is able to locate JavaScript fragments carried out in URLs. Finally, we propose a framework for user-tracking in every-day Internet activities, that can protect users from data fabrication caused by malicious ISPs.

1.2 Novelty and Contributions

In this dissertation we make numerous contributions. More precisely, the contributions are the following:

1. We identify *return-to-JavaScript* attacks; a new XSS flavor, which can defeat web applications that use script whitelisting [JSH07] to counter code injections. We discuss return-to-JavaScript attacks in Chapter 3.

[1]The form "whitelist" is used instead of "white-list", due to the massive adoption of the first form by the community.

2. We design, implement and evaluate xJS and RaJa; two frameworks, which apply ISR concepts to web applications. We discuss xJS and RaJa in Chapter 4 and 5.

3. We examine the code base of four real-world web applications and we show that with minimal modifications all legitimate JavaScript can be efficiently isolated in a machine-driven way. This is crucial for all applications that aim at applying operations in the JavaScript part of a web application. We discuss JavaScript code-mixing and isolation in Chapter 5.

4. We design, implement and evaluate a network-based monitoring prototype that can efficiently locate JavaScript fragments carried-out in URLs. We discuss xHunter in Chapter 6.

5. We present an analysis of a large set of about 12,000 real-world web exploits collected by a well known repository, XSSed.com [FP]. We discuss this analysis in Chapter 7.

6. We design, implement and evaluate Network Flow Contracts (NFCs), a framework against data fabrication at the ISP's level. An ISP providing NFCs can guarantee that all network activity has been logged with the subscriber's verification. We discuss NFCs in Chapter 8.

1.3 Dissertation Organization

This dissertation is organized as follows. In Chapter 2 a short introduction of background knowledge is presented. This chapter is essential for understanding the rest of the content. A new flavor of XSS exploitation, namely *return-to-JavaScript* attacks, that can bypass script whitelisting, is presented in Chapter 3. In Chapter 4 and 5 the architecture and the design of xJS and RaJa are presented respectively. Both frameworks consider ISR as a mean for protecting web applications from code injections. In Chapter 6 a network-level XSS detector is described, namely xHunter. With the assistance of xHunter, a large case study of about 12,000 XSS exploits hosted in a real-world repository is presented in Chapter 7. Network Flow Contracts are presented and evaluated in Chapter 8. Related work is listed in Chapter 9.

This dissertation concludes in Chapter 10. A list of publications related to this dissertation's content is presented in Appendix A.

Chapter 2

Background

In this chapter we give a short introduction of some important concepts used in the rest of this dissertation's content.

2.1 The Web2.0 Era

The web, as we know it, dates back to early 90s. The basic technologies associated with the web have not changed through the years, although some of them have heavily evolved. The term *Web2.0* was invented sometime between late 90s and early 2000. It is hard to give a precise definition for Web2.0. However, it is considered that the major shift from traditional web pages to Web2.0 applications was due to the introduction of `XMLHttpRequest`, a JavaScript function implemented by all modern web browsers, which is able to perform an HTTP request asynchronously. A web page can update the contents of a very precisely defined region by performing an HTTP request in the web server using an `XMLHttpRequest` call. This partial update gives the notion of interaction. Using partial updates, a web page can be enhanced with pull-down menus, buttons, actions and other forms of user interface elements. Thus, the web page is transformed to an application, which resembles the behavior of a desktop application. There are many libraries in the market that use `XMLHttpRequest` as the basic building block for constructing rich client-side interfaces. Very frequently we refer to this libraries with the term Asynchronous JavaScript and XML (AJAX) [G+05]. Among popular web applications built using rich AJAX interfaces, some examples are Gmail, a full featured e-mail client provided by Google, GoogleDocs, a full featured office suite provided by Google, Flickr, a photograph manager owned by Yahoo, and Facebook, a social network utility with more than 500 millions of registered users.

The invention of AJAX gave significant boost to the web's usage. In parallel, the complexity and the sophistication of modern web browsers transformed the web platform into a rich media for building easy-to-use applications. More importantly, what many desktop applications lack, compared to web applications is the easiness of adding collaboration features. A web application, by definition, provides content sharing among users and makes collaboration easy. Many web applications, like Facebook for example, have formed a huge user base, which interacts with content using the same platform. [1] Moreover, the transition from static web pages to full featured web applications increased significantly the source code. Consider that a typical web application, like Gmail for example, is composed by tens of thousands lines of code. This, inevitably, introduces bugs, which can be leveraged as vulnerabilities, which in turn can be exploited to compromise web applications as it is the case for native applications. We now give a short review of well-known methods for compromising web applications.

2.2 Security Threats in Web Applications

According to a September-2009 report published by the SANS Institute [SAN], *attacks against web applications constitute more than 60% of the total attack attempts observed on the Internet. Web application vulnerabilities such as SQL injection and Cross-Site Scripting flaws in open-source as well as custom-built applications account for more than 80% of the vulnerabilities being discovered.* The majority of web attacks are based in code injections, more frequently occurring by the injection of some JavaScript code in a web application. However, there are many web attack variants. We shortly discuss the most well known.

[1]We have shown that this can be abused for malicious purposes [LAAA06, AMA$^+$08, MAA$^+$10].

2.2.1 Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS) is considered as the most severe variant of a web attack. A web application suffering of XSS has a great probability to become fully compromised. An XSS takes place when an attacker injects JavaScript code in a web application that expects some data input. The server fails to filter out the data input and treats it as code. The result is that the attacker's code will be eventually executed in a web browser.

Now, assume that the attacker's code is set to get the currently loaded cookie of a user's session and post it to a server controlled by the attacker. This code will be rendered by every web browser, which comes in contact with the vulnerable web application. When the code starts executing in the web browser, the current cookie will be the user's cookie, who is viewing the web application. Thus, the final result is for the attacker to collect all cookies, associated with users interacting with the web application.

There are many different ways to inject malicious content in a web application. More often, an attacker tries to inject JavaScript, but damage can be also caused by injecting an `iframe` or an image.

2.2.2 Cross-Site Request Forgery (CSRF)

Another popular web attack flavor, not as powerful as XSS but being able to become severe for the end-user, is Cross-Site Request Forgery (CSRF) [BJM08]. During a CSRF attack, the attacker aims at forcing a victim's web browser to artificially generate requests towards a web application. In the case that the victim is logged in the target web application, then the artificially created requests will be landed at the web application with the victim's credentials.

Consider, for example, Alice while using Gmail, she opens the attacker's web site in a new tab. Upon visiting the attacker's web server, Alice's browser is forced to generate a request that deletes a Gmail inbox. Alice is using Gmail and thus all requests will leave Alice's web browser with her credentials. Thus, Alice's inbox will be deleted. Notice, that we assume that Alice maintains an open connection with the target web application, while she is visiting the attacker's web page.

2.2.3 Visual Attacks

A family of web attacks is based on deceiving users by creating specially crafted visual conditions. Representative attacks of this category are Clickjacking [HG08] and Phishing [DTH06]. In Clickjacking, the attacker is using overlapped layers to hide from the user her real actions. For example, consider a dialog box with two buttons, one confirming the action and one denying it. Consider, now, that the attacker has injected a visual layer, over the dialog box, which visually exchanges the two buttons. Thus, the user finally receives the dialog box with the buttons exchanged. If the user wants to deny the action, she will actually click on the button, which confirms the action. However, the user is not able to know this in advance, since the confirmation button is hidden under the visual representation of the deny button. Clickjacking works, because a special visual property of HTML elements, called the *z-index*, makes easy to create piles of graphic areas.

On the other hand, Phishing aims at deceiving a user by creating entire clones of actual web sites and web applications. An attacker creates a web site, which looks identical with a well-known web application, for example Gmail. The content of the application is served and controlled by the attacker's web server. The attacker tricks the user to enter her credentials in the cloned web site. The user proceeds and does so, since she believes that she is logging in the authentic web site. The attacker collects the victim's credentials and uses them to enter the authentic web site.

2.2.4 Other

Lately, there have been many research efforts towards the invention of very elaborate and sophisticated web attacks. Apart from XSS and CSRF, there are many ways to exploit a web application. For example, by sending specifically crafted data, someone can exploit bugs in an AJAX-based client [SHPS] or by including web pages as a Cascade Style Sheet (CSS) someone can steal a user's secret [LSZCC10] (for example the subjects of her web inbox) from a web application. Moreover, even devices such as routers that use web interfaces for configuration can be exploited by injecting specific client-side code [BBB09]. All these flavors of web exploitation can in principle use JavaScript. In the context of this dissertation, we do not attempt to deal with all of these problems. However, if the attack is based on JavaScript usage, then the frameworks we propose can successfully detect and prevent exploitation.

2.3 Instruction Set Randomization

Instruction Set Randomization (ISR) is a generic technique for countering code injections. The fundamental concept of ISR is to randomize the instruction set of a particular executing flow, so that every time the flow runs the commands are different. Thus, any code injection is hard to glue in the rest of the program's code, unless the key for *de-randomizing* every instant is known. As it is described by Angelos D. Keromytis: *The basic idea behind this approach is that attackers don't know the language "spoken" by the runtime environment on which an application runs, so a code-injection attack will ultimately fail because the foreign code, however injected, is written in a different language. In contrast to other defense mechanisms, we can apply ISR against any type of code-injection attack, in any environment.* [Ker09]

ISR has been applied in native applications [KKP03] and in SQL [BK04]. Some concepts of ISR has been applied to counter XSS in Noncespaces [GC09], where a randomized DOM [LHLHW^{+}04] is introduced. Some discussion about applying ISR concepts for XSS can be found in [Ker09]. A significant part of this dissertation is devoted in two frameworks, xJS and RaJa , which introduce ISR in web applications.

Chapter 3

Return To JavaScript Attacks

In this chapter we present a new flavor of XSS, which we refer to it as *return-to-JavaScript* attack, in analogy with the *return-to-libc* attack in native code. This kind of XSS attack can escape script whitelisting, used by existing XSS mitigation schemes. We further highlight some important issues for DOM-based XSS mitigation schemes. All the attacks listed in this section can be successfully prevented by xJS and RaJa, which are discussed in Chapters 4 and 5.

3.1 Overview

A practical mitigation scheme for XSS attacks is script whitelisting, proposed in BEEP[JSH07]. BEEP works as follows. The web application includes a list of cryptographic hashes of valid (trusted) client-side scripts. The browser, using a *hook*, checks upon execution of a script if there is a cryptographic hash in the whitelist. If the hash is found, the script is considered trusted and executed by the browser. If not, the script is considered non-trusted and the policy defines whether the script may be rendered or not. *Script whitelisting is not sufficient.* Despite its novelty, we argue here that simple whitelisting may not prove to be a sufficient countermeasure against XSS attacks. To this end, consider the following.

Location of trusted scripts. As a first example, note that BEEP does not examine the script's location inside the web document. Consider the simple case where an attacker injects a trusted script, initially configured to run upon a user's click (using the `onclick` action), to be rendered upon document loading (using the `onload`[1] action). In this case, the script will be executed, since it is already whitelisted, but not as intended by the original design of the site; the script will be executed upon site loading and not following a user's click. If, for example, the script deletes data, then the data will be erased when the user's browser loads the web document and not when the user clicks on the associated hyperlink.

Exploiting legitimate whitelisted code. Attacks may be further carried out through legitimate whitelisted code. XSS attacks are typically associated with injecting arbitrary client-side code in a web document, which is assumed to be foreign, i.e., not generated by the web server. However, it is possible to perform an XSS attack by placing code that *is* generated by the web server in different regions of the web page. This attack resembles the classic *return-to-libc* attack [Des97] in native applications and thus we refer to as *return-to-JavaScript*. Return-oriented programming suggests that an exploit may simply transfer execution to a place in `libc`[2], which may cause again execution of arbitrary code on behalf of the attacker. The difference with the traditional buffer overflow attack [One96] is that the attacker has not injected any *foreign* code in the program. Instead, she transfers execution to a point that already hosts code that can assist her goal. A similar approach can be used by an attacker to escape whitelisting in the web environment. Instead of injecting her own code, she can take advantage of existing *whitelisted* code available in the web application. Note that, typically, a large fraction of client-side code is not executed upon document loading, but is triggered during user events, such as mouse clicks. Below we enumerate some possible scenarios for XSS attacks based on whitelisted code, which can produce (i) annoyance, (ii) data loss and (iii) complete takeover of a web site.

Annoyance. Assume the blog site shown in Figure 3.1. The blog contains a JavaScript function `logout()`, which is executed when the user clicks the corresponding hyperlink, *Logout*

[1]One can argue that the `onload` action is limited and usually associated with the `<body>` tag. The latter is considered hard to be altered through a code-injection attack. However, note, that the `onload` event is also available for other elements (e.g. images, using the `<img>` tag) included in the web document.

[2]This can also happen with other libraries as well, but `libc` seems ideal since (a) it is linked to every program and (b) it supports operations like `system()`, `exec()`, `adduser()`, etc., which can be (ab)used accordingly. More interestingly, the attack can happen with no function calls but using available combinations of existing code [Sha07].

```
1: <html>
2:  <head> <title> Blog! </title> <head>
3:  <body>
4:   <a onclick="logout();">Logout</a>
5:   <div class="blog_entry" id="123"> {...} <input type="button" onclick="delete(123)"></div>
6:   <div class="blog_comments"> <ul>
7:    <li> <img onload="logout();" src="logo.gif">
8:    <li> <img onload="window.location.href='http://www.google.com';" src="logo.gif">
9:    <li> <img onload="delete(123);">
10:  </div>
11:  <a onclick="window.location.href='http://www.google.com';">Google</a>
12: </body>
13:</html>
```

Figure 3.1: A minimal Blog site demonstrating the whitelisting attacks.

(line 4 in Fig. 3.1). An attacker could perform an XSS attack by placing a script that calls `logout()` when a blog entry is rendered (see line 7 in Fig. 3.1). Hence, a user reading the blog story will be forced to logout. In a similar fashion, a web site that uses JavaScript code to perform redirection (for example using `window.location.href = new-site`) can be also attacked by placing this whitelisted code in an `onload` event (see line 8 in Fig. 3.1).

Data Loss. A web site hosting user content that can be deleted using client-side code can be attacked by injecting the whitelisted deletion code in an `onload` event (see line 9 in Fig. 3.1). AJAX [G$^+$05] interfaces are popular in social networks such as Facebook.com and MySpace.com. This attack can be considered similar to a SQL injection [Anl02], since the attacker is implicitly granted access to the web site's database.

Complete Takeover. Theoretically, a web site that has a full featured AJAX interface can be completely taken over, since the attacker has all the functionality she needs a-priori whitelisted by the web server. For example, an e-banking site that uses a JavaScript `transact()` function for all the user transactions is vulnerable to XSS attacks that perform arbitrary transactions.

A workaround to mitigate the attacks presented above is to include the event type during the whitelisting process. Upon execution of a function, such as `logout()` (see Figure 3.1), which is triggered by an `onclick` event, the browser should check the whitelist for finding a hash key for `logout()` *associated with an* `onclick` *event.* However, this can mitigate attacks which are based on using existing code with a different event type than the one initially intended to by the web programmer. Attacks may still happen. Consider the *Data*

Loss scenario described above, where an attacker places the deletion code in `onclick` events associated with new web document's regions. The attacker achieves to execute legitimate code upon an event which is not initially scheduled. Although the attacker has not injected her own code, she manages to escape the web site's logic and to associate legitimate code with her own actions. Attacks against whitelisting, based on injecting malicious data in whitelisted scripts, rather than injecting whitelisted code, have been described in [NSS09].

3.2 DOM-based Attacks

There is a number of proposals [GC09, NSS09, FHEW08] against XSS attacks, which are based on information and features provided by the Document Object Model (DOM) [LHLHW$^+$04]. Every web document is rendered according to DOM, which represents essentially its esoteric structure. This structure can be utilized in order to detect or prevent XSS attacks. One of the most prominent and early published DOM-based techniques is DOM sandboxing, introduced originally in BEEP.

DOM sandboxing works as follows. The web server places all scripts inside `div` or `span` HTML elements that are attributed as *trusted*. The web browser, upon rendering, parses the DOM tree and executes client-side scripts only when they are contained in *trusted* DOM elements. All other scripts are marked as non-trusted and they are treated according to the policies defined by the web server. We discuss here in detail three major weaknesses of DOM sanbdoxing as an XSS mitigation scheme: (i) node splitting, (ii) element annotation and (iii) DOM presence.

Node splitting. An attacker can inject a malicious script surrounded, on purpose, by misplaced HTML tags in order to escape from a particular DOM node. Consider for example the construct:

```
<i>{ message }<\i>
```

which, denotes that a message should be rendered in *italic* style. If the message variable is filled in with:

```
</i><b> bold message </b><i>
```

then the carefully placed `<i>` and `<b>` tags result the message to be displayed in **bold** style, rather than *italic*.

The authors of BEEP suggest a workaround for dealing with node-splitting. All the data inside an untrusted `div` must be placed using a special coding idiom in JavaScript. A more elegant approach is to randomize the DOM elements [GC09, NSS09]. The attacker then needs to know a key in order to escape from a node. Consider the example:

```
<nonce42:div> ... </nonce42:div>
```

In this case the key is 'nonce42'. As long as the key is complex enough to be predicted through a brute force attack, this technique is considered sufficient. However, the only published implementation so far, Noncespaces, is based on XHTML [PAA$^+$00] which is a strict HTML dialect. XHTML inherits many properties from XML. Currently, an XHTML document must comply to various criterions of validness in order to be rendered by a browser. Noncespaces is based on this strictness to identify potential attacks. Any code-injection attempt produces a non-valid document, which is rejected by the browser (i.e., the page is not rendered). An attacker may exploit this strictness by injecting malformed HTML code on purpose, invalidating this way the target web pages. For a practical example of this attack, consider a malicious user that posts malformed HTML messages in users' profiles of a social network.

Element annotation. Enforcing selective execution in certain areas of a web page requires identification of those DOM elements that may host untrusted code or parts of the web application's code that inject unsafe content. This identification process is far from trivial, since the complexity of modern web pages is high, and web applications are nowadays composed of thousands lines of code. To support this, in Table 3.1 we highlight the number of `script`, `div` and `span` elements of a few representative web page samples. Such elements can be in the order of thousands in modern web pages. While there is active research to automate the process of marking untrusted data [Sek09, LC06] or to discover taint-style vulnerabilities [JKK06, ML08], we believe that, currently, the overhead of element annotation is prohibitive, and requires, at least partially, human intervention.

DOM presence. All DOM-based solutions require the presence of a DOM tree. However,

	Facebook.com	MySpace.com	Digg.com
`script`	23	93	82
`div`	2,708	264	302
`span`	982	91	156

Table 3.1: Element counts of popular home pages indicating their complexity.

XSS attacks do not always require a DOM tree to take place. For example, consider an XSS attack which bypasses the content-sniffing algorithm of a browser and is *carried within* a PostScript file [BCS09]. The attack will be launched when the file is previewed, and there is high probability that upon previewing there will be no DOM tree to surround the injected code. As browsers have been transformed to a generic preview tool, we believe that variants of this attack will manifest in the near future.

Another example is the unofficially termed *DOM-Based XSS* or *XSS of the Third Kind* attacks [Kle]. This XSS type alters the DOM tree of an already rendered page. The malicious XSS code does not interact with the server in any way. In such an attack, the malicious code is embedded inside a URI after the fragment identifier. [3] This means that the malicious code (a) is not part of the initial DOM tree and (b) is never transmitted to the server. Unavoidably, DOM-based solutions [GC09, NSS09] that define trust classes in the DOM tree at server side will fail. The exploit will never reach the server and, thus, never be associated with or contained in a trust class.

[3] For more details about the fragment identifier, we refer the reader to `http://www.w3.org/DesignIssues/Fragment.html`.

Chapter 4

Countering XSS Using Isolation Operators

xJS is a complete framework for the detection and prevention of XSS attacks in web applications. Compared to similar frameworks [JSH07], xJS is easy to implement, can be deployed in a backwards compatible fashion, has very low computational overhead and can cope with return-to-JavaScript attacks.

4.1 Overview

Several studies have proposed mechanisms and architectures based on policies, communicated from the web server to the web browser, to mitigate XSS attacks. The current state of the art includes XSS mitigation schemes proposing whitelisting of legitimate scripts [JSH07], utilizing randomized XML namespaces for applying trust classes in the DOM [GC09], or detecting code injections by examining modifications to a web document's original DOM structure [NSS09]. While we believe that the aforementioned techniques are promising and in the right direction, they have weaknesses and they fail in a number of cases. As we discussed in Chapter 3, whitelisting fails to protect from attacks that are based on already whitelisted scripts, while DOM-based solutions fail to protect from attacks where the DOM tree is absent [BCS09].

To account for these weaknesses, we propose xJS, which is a practical and simple framework that isolates legitimate client-side code from any possible code injection. The framework can be seen as a *fast randomization* technique. Instruction Set Randomization (ISR) [KKP03] has been proposed for defending against code injections in native code or in other environments, such as code executed by databases [BK04]. However, we believe that adapting ISR to deal with XSS attacks is not trivial. This is because *web client-side code* is produced by the server and is executed in the client; the server lacks all needed functionality to manipulate the produced code. For example, randomizing the JavaScript instruction set in the web server requires at least one full JavaScript parser running at the server. Thus, instead of blindly implementing ISR for JavaScript, our design introduces *Isolation Operators*, which transpose all produced code in a new isolated domain. In our case, this is the domain defined by the `XOR` operator.

We design xJS with two main properties in mind:

- **Backwards Compatibility.** We aim for a practical, developer-friendly solution for constructing secure web applications and we ensure that the scheme provides backwards compatibility. xJS allows web servers to communicate to web browsers when the scheme is enabled or not. A web browser not supporting the framework may still render web applications, albeit without providing any of the security guarantees of xJS.

- **Low Computation Overhead**. Our design avoids the additional overhead of applying ISR in both web server and client, which would significantly increase the computational overheads. This is because the web code would be parsed twice (one in the server during serving and one in the client during execution). Instead, the isolation operator introduced in xJS applies the `XOR` function to the whole source corpus of all legitimate client-side code. Thus, the randomization process is fast, since XOR exists as a CPU instruction in all modern hardware platforms, and does not depend on any particular instruction set.

We implement and evaluate our solution in FireFox and WebKit[1], and in the Apache web server.

[1]WebKit is not a web browser itself, it is more like an application framework that provides a foundation upon which to build a web browser. We evaluate our modifications on WebKit using the Safari web browser.

```
<div>                         <div>
<img onload=''render();''>    <img onload=''AlCtV...''>
<script>                      <script>
alert(''Hello World'');         vpSUlJTV2NHGwJyW/NHY...
<script>                      </script>
</div>                        </div>
```

Figure 4.1: Example of a web page that is generated by our framework. On the left, the figure shows the source code as exists in the web server and on the right the same source as it is fetched by the web browser. The JavaScript source has been `XOR`ed and a Base64 encoding has been applied in order to transpose all non-printable characters to the printable ASCII range.

Our evaluation shows that xJS can successfully prevent *all* 1,380 attacks of a well-known repository [FP]. At the same time, it imposes negligible computational overhead in the server and in the client side. Finally, our modifications appear to have no negative side-effects in the user web browsing experience. To examine user-perceived performance, we examine the behavior of xJS-enabled browsers through a leading JavaScript benchmark suite [sun], which produces the same performance results in both the xJS-enabled and the original web browsers.

4.1.1 Attacks Not Addressed

xJS aims on protecting against XSS attacks that are based on JavaScript injections. The framework is not designed for providing defenses against `iframe` injections and drive-by downloads [PMRM08], injections that are non-JavaScript based (for example, through arguments passed in Flash objects) and Phishing [DTH06]. However, some fundamental concepts of xJS can be possibly applied to non-JavaScript injections.

4.2 Architecture

The fundamental concepts of our framework are *Isolation Operators* and *Action Based Policies* in the browser environment. We review each of these concepts in this section and, finally, we provide information about our implementation prototypes. xJS is a framework that can address XSS attacks carried out through JavaScript. However, our basic concept can be also applied to other client-side technologies, such as Adobe Flash. The basic properties of the proposed framework can be summarized in the following points.

- xJS prevents JavaScript code injections that are based on third party code or on code that is already used by the trusted web site.
- xJS prevents execution of trusted code during an event that is not scheduled for execution. Our framework guarantees that *only* the web site's code will be executed and *only* as the site's logic defines it.
- xJS allows for multiple trust-levels depending on desired policies. Thus, through xJS, parts of a web page may require elevated trust levels or further user authentication to be executed.
- xJS in principle prevents attacks that are based on injected data and misuse of the JavaScript `eval()` function. We discuss `eval()` semantics in detail in Sections 4.4 and 4.5.

4.2.1 Isolation Operators

xJS is based on Instruction Set Randomization (ISR), which has been applied to native code [KKP03] and to SQL [BK04]. The basic concept behind ISR is to randomize the instruction set in such a way so that a code injection is not able to *speak the language of the environment* [Ker09] and thus is not able to execute. In xJS, inspired by ISR, we introduce the concept of Isolation Operators (IO). An IO essentially transposes a source corpus to a new isolated domain. In order to de-isolate the source from the isolated domain a unique key is needed. This way, the whole source corpus, and not just the instruction set, is randomized.

Based on the above discussion, the basic operation of xJS is the following. We apply an IO such as the `XOR` function to effectively randomize and thus isolate all JavaScript source of a web page. The isolation is achieved since all code has been transposed to a new domain: the `XOR` domain. The IO is applied by the web server and all documents are served in their isolated form. To render the page, the web browser has to *de-isolate* the source by applying again the IO and then execute it.

Note that, in xJS, we follow the approach of randomizing the whole source corpus and not just the instruction set as in the basic ISR concept. We proceed with this choice since the web code is produced in the web server and it is executed in the web browser. In addition, the server lacks all needed functionality to manipulate the produced code, since it is not JavaScript-aware. For example, randomizing the JavaScript instruction set needs at least one full JavaScript parser running at the server. This can significantly increase the computational overhead and user-perceived latency, since the code would be parsed twice (one in the server during serving and one in the client during execution). However, the isolation can break web applications that explicitly evaluate dynamic JavaScript code using `eval()`. In that case, the web developer must use a new API, `xeval()`, since xJS alters the semantics of `eval()`. We further discuss this in Section 4.5. Finally, we select `XOR` as the IO because it is in general considered a fast process; all modern hardware platforms include a native implementation of the `XOR` function. However, our framework may be applied with any other IO.

Figure 4.1 depicts an xJS example. On the left, we show the source code as it exists in the web server and on the right, we provide the same source as it is fetched by the web browser. The JavaScript source has been `XOR`ed and a Base64 [Jos06] encoding has been applied in order to transpose all non-printable characters to the printable ASCII range.

4.2.2 Action Based Policies

xJS allows for multiple trust-levels for the same web site depending on the desired operation. In general, our framework suggests that policies should be expressed as actions. Essentially, all trusted code should be treated using the policy "*de-isolate and execute*". For different trust levels, multiple IOs can be used or the same IO can be applied with a different key. For example, portions of client-side code can be marked with different trust levels. Each portion will be isolated using the `XOR` function, but with a different key. The keys are transmitted in HTTP headers (see the use of `X-IO-Key`, later in this section) every time the server sends the page to the browser.

Expressing the policies in terms of actions has the following benefit. The injected code cannot bypass the policy, unless it manages to produce the needed result after the action is applied to it. The latter is considered practically very hard, even for trivial actions such as the XOR operation. One possible direction for escaping the policy is using a brute force attack. However, if the key is large enough the probability of a brute force attack to succeed is low.

Defining the desired policy set is out of the scope of this thesis. For the purpose of our evaluation (see Section 4.4) we use one policy, which is expressed as "*de-isolate (apply XOR) and execute*". Other example policies can be expressed as "de-isolate and execute under user confirmation", "de-isolate with the X key and execute", etc.

4.3 Implementation

We implemented xJS by modifying Firefox and WebKit. We also created a module for the Apache web server.

Browser Modifications. All three modified web browsers operate in the following way. A custom HTTP header field, `X-IO-Key`, is identified in each HTTP response. If the key is present, this is an indication that the web server supports the framework, and the field's value denotes the key for the de-isolation process. This is also a practical way for incremental deployment of the framework in a backwards compatible fashion. At the moment, we do not support multiple keys, but extending the browser with such a feature is considered trivial. On the other hand, the web browser communicates to the web server that it supports the framework using an `Accept`[2] header field for every HTTP request.

As far as WebKit is concerned, we had to modify two separate functions. First, the function that handles all events (such as `onload`, `onclick`, etc.), and second, the function that evaluates a JavaScript code block. We modified these functions to (i) decode all source using Base64 and (ii) apply the `XOR` operation with the de-isolation key (the one transmitted in `X-IO-Key`) to each byte. FireFox has a different design. It also uses two functions, one for compiling a JavaScript function and one for compiling a script. However, these functions operate recursively. We further discuss this issue in Section 4.4. In both, Firefox and WebKit, the modifications do not exceed 600 lines of code.

[2]For the definition of the `Accept` field in HTTP requests, see: `http://www.w3.org/Protocols/HTTP/HTRQ_Headers.html#z3`

Server Modifications. For the server part of xJS we are taking advantage of the modular architecture of the Apache web server. During Apache's start-up phase all configuration files are parsed and modules that are concerned with processing an HTTP request are loaded. The main processing unit of the apache web server is the content generator module. A module can register content generators by defining a handler that is configurable by using the `SetHandler` or `AddHandler` directives. These can be found in Apache's configuration file (httpd.conf).

Various request phases that precede the content generator exist. They are used to examine and possibly manipulate some request headers, or to determine how the request will be handled. For example the request URL will be matched against the configuration, because a certain content generator must be used. In addition the request URL may be mapped to a static file, a CGI script or a dynamic document according to the content generator's operation. Finally after the content generator has sent a reply to the browser, Apache logs the request.

Apache (from version 2 and above) also supports filters. Consider the filter chain as a data axis, orthogonal to the request processing axis. The request data may be processed by input filters before reaching the content generator. After the generator has finished generating the response various output filters may process it before being sent to the browser. We have created an Apache module which operates as a content generator. For every request, that corresponds to an HTML file in the disk, the file is fetched and processed by our module. The file is loaded in memory and stored in a buffer. The buffer is transfered to an HTML parser (based on the `HTMLParser` module from libxml2 [Vei04]). This is an HTML 4.0 non-verifying parser with API compatible with the XML parser ones. When the parsing is done our module traverses the parser's XML nodes in memory and searches for all nodes that contain JavaScript (`<script>` nodes and events). If there is a match the XOR operation is applied using the isolation key to each byte of the JavaScript source. Finally all source is encoded in Base64.

After encoding all possible JavaScript source in the web page, the buffer is sent to the next operating module in the chain; this might be an output filter or the web browser. Implementing xJS as a content generator module has the benefit of isolating by encryption all

JavaScript source before any dynamic content, which might include XSS attacks, is inserted. Our framework can cooperate with other server-side technologies, such as PHP, in two ways: (a) by using two Apache's servers (one running xJS and the other one the PHP module) and (b) by configuring PHP to run as a filter. All evaluation results presented in Section 4.4 are collected using the second setup.

Secret Key. The secret key that is used for the `XOR` operation is a string of random alphanumeric characters. The length of the string can be arbitrary. For all experiments presented a two-character string is used. Assuming that S_l is the JavaScript source of length l and K_L is the secret key of length L, the encoding works as follows: $Enc(S_i) = S_i \oplus K_{(i \,\%\, L)}, 0 < i < l$. It is implied that the ASCII values of the characters are used. The secret key is refreshed per request. We do not consider Man-in-the-Middle (MiM) attacks, since during a MiM an attacker can alter the whole JavaScript source without the need of an injection through XSS.

4.4 Evaluation

In this section we evaluate the xJS prototype. Our evaluation seeks to answer four questions: (a) how many real XSS attacks can be prevented, (b) what the overhead on the server is, (c) what the overhead on the web browser is and, finally, (d) whether the framework imposes any side-effects in the user's browsing experience.

4.4.1 Attack Coverage

We first evaluate the effectiveness of the xJS framework to prevent real-world XSS attacks. xJS aims on preventing traditional XSS attacks, as well as the XSS attacks described in Chapter 3.

Real-world exploits. To verify that xJS can cope with real-world XSS exploits, we use the repository hosted by XSSed.com [FP] which includes a few thousands of XSS vulnerable web pages. This repository has been also used for evaluation in other papers [NSS09]. The evaluation of the attack coverage through the repository is not a straightforward process.

First, XSSed.com mirrors all vulnerable web pages with the XSS code embedded in their body. Some of them have been fixed after the publication of the vulnerability. These updated pages cannot be of use, since xJS prevents the code injection before it takes place and there is no way for us to have a copy of the original vulnerable web page (without the XSS code in its body). Second, we have no access to the vulnerable web server and, thus, we cannot use our server-side filter for the evaluation.

To address the aforementioned limitations, we conduct the evaluation as follows. First, we resolve all web sites that are still vulnerable. To this end, we download all 10,154 web pages listed in XSSed.com, along with their attack vectors. As the attack vector we define the URL along with the parameters that trigger the vulnerability.[3] Since XSS attacks that are based on a redirection without using any JavaScript cannot be addressed by xJS, we remove all such cases. Thus, we exclude 384 URLs that have an `iframe` as attack vector, 416 URLs that have a redirection to XSSed.com as attack vector and 60 URLs that have both an `iframe` and a redirection to XSSed.com as attack vector.

After this first pre-processing stage, the URL set contains all web pages that were vulnerable at some period in time and their vulnerability can be triggered using JavaScript; for example, the attack vector contains a call to the `alert()` function. We then exclude from the set all web-pages for which their vulnerability has been fixed after it became public in XSSed.com. To achieve this, we request each potentially vulnerable page through a custom proxy server we built using BeautifulSoup [Ric08]. The task of the proxy is to attach some JavaScript code that overrides the `alert()` function with a URL request to a web server located in our local network. Since all attack vectors are based on the `alert()` function the web server recorded all successful attacks in its access logs. Using this methodology we manage to identify 1,381 web pages which are still vulnerable as of early September 2009. Our methodology suggests that about 1 in 9 web pages have not been fixed even after the vulnerability was published.

We use the remaining 1,381 pages as our final testing set. Since we cannot install our modified Apache in each of the vulnerable web sites, we use our proxy for simulating the

[3]For example, consider the attack vector: `http://www.needforspeed.com/undercover/home.action?lang=\"><script>alert(document.cookie);</script>&region=us`

server-side portion of xJS. More precisely, for each vulnerable page, we request the vulnerable document through our proxy with a slightly altered vector. For example, for the following attack vector,

```
http://site.com/page?id=<script>alert("XSS");</script>
```

the proxy instead requests the URL,

```
http://site.com/page?id=<xscript>alert("XSS");</xscript>.
```

Notice that the `script` tag has been modified to `xscript`. Using this methodology, we manage to build all vulnerable web pages with *the attack vector embedded but not in effect.* However, the JavaScript code contained in the web document is not isolated. Thus, the next step is to force the proxy to parse all web documents and apply the XOR function to the JavaScript code. At this point, all vulnerable web pages have the JavaScript code isolated and the attack vector defunct. Hence, the last step is to re-enable the attack vector by replacing the `xscript` with `script` and return the web page to the browser. All web pages also include some JavaScript code responsible for the `alert()` overloading. This code modifies all `alert()` calls to perform a web request to a web server hosted in our local network. If our web server records requests, the `alert()` function is called or, in other words, the XSS exploit run.

To summarize the above process, our experiment to evaluate the efficacy of the xJS framework is the following. We request each web page from the collected set which includes 1,381 still vulnerable web pages through a custom proxy that performs all actions described above. All web pages are requested using a modified Firefox. We select the modified Firefox in Linux, because it is easier to instrument through a script. We manually tested a random sample of attacks with a modified version of WebKit and we recorded identical behavior.

After Firefox has requested all 1,381 vulnerable pages through our custom proxy, we inspect our web server's logs to see if any of the XSS attacks succeeded. Our web server recorded only one attack. We carefully examined manually this particular attack and found out that it is a web page that has the XSS exploit stored inside its body and not in its attack vector [xssb]. The particular attack succeeded just as a side-effect of our evaluation methodology. If xJS was deployed in the vulnerable web server, this particular attack would also have been prevented. Hence, *all* 1,380 real-world XSS attacks were prevented successfully by our framework.

Attacks presented in Chapter 3. For the attacks presented in Chapter 3, since to our knowledge they have not been observed in the wild yet, we performed various custom attack scenarios using a popular web framework, Ruby on Rails [TH06]. We created a vulnerable blog and then installed the vulnerable blog service to a modified Apache server and browsed the blog using the modified web browsers. As expected, in all cases, xJS succeeded in preventing the attacks.

We now look at specific attacks such as the ones based on a code injection in data and the careless use of `eval()`. The injected code is in plain text (non-isolated), but unfortunately it is attached to the isolated code after the de-isolation process. The injected code will be executed as if it is trusted. However, there is a way to prevent this. In fact, the internal design of Firefox gives us this feature with no extra cost. Firefox uses a `js_CompileScript()` function in order to compile JavaScript code. The design of this function is recursive and it is essentially the implementation of the actual `eval()` function of JavaScript. When Firefox identifies the script `eval($_GET('id'));`, de-isolates it, calls the `eval()` function, which in principle calls itself in order to execute the `$_GET('id')` part. At the second call, the `eval()` again de-isolates the `$_GET('id')` code, which is in plain text. The second de-isolation process fails and thus the code does not execute.

Our Firefox implementation can address this type of attack. WebKit must be further modified to support this functionality. However, this modification affects the semantics of `eval()`. For a more detailed discussion, please see Section 4.5.

4.4.2 Server Overhead

We now measure the overhead imposed on the server by xJS. To this end, we request a set of web pages that embed a significant amount of JavaScript. We choose to use the SunSpider suite [sun] for this purpose. The SunSpider suite is a collection of JavaScript benchmarks that ship with WebKit and measure the performance of JavaScript engines. It is composed of nine different groups of programs that perform various complex operations. We manually select

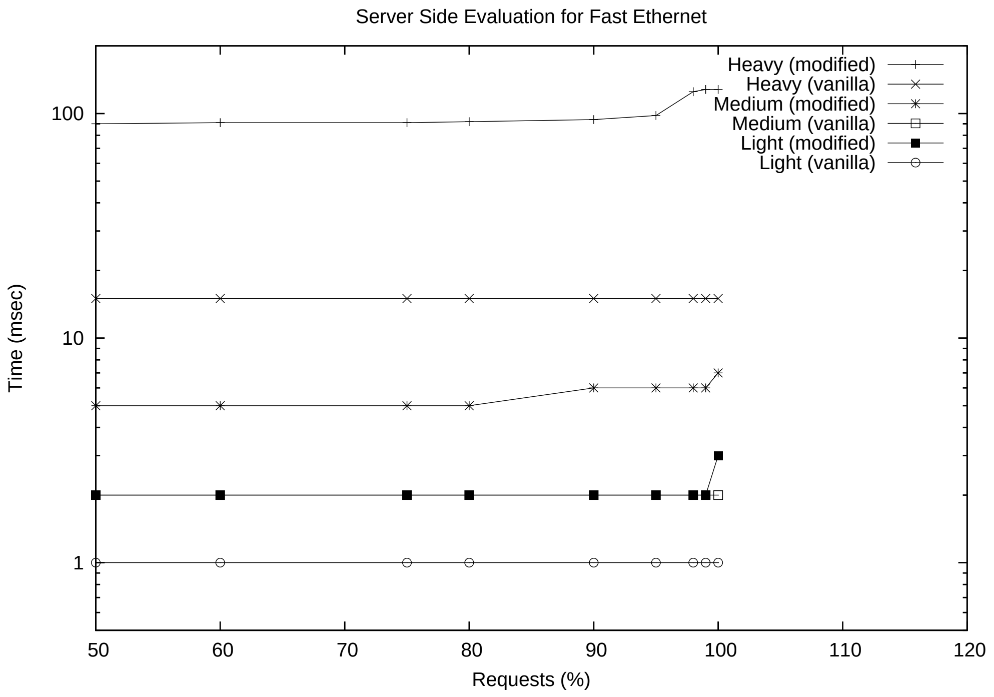


Figure 4.2: Server side evaluation when the Apache benchmark tool (`ab`) is requesting each web page through a Fast Ethernet link. In the worst case (heavy) the server imposes delay of a factor of five greater, while in the normal case the delay is only a few milliseconds.

three JavaScript tests from the SunSpider suite. The *heavy* test involves string operations with many lines of JavaScript. This is probably the most processing-intensive test in the whole suite, composed of many lines of code. The *normal* test includes a typical amount of source code like most other tests that are part of the suite. Finally, the *light* test includes only a few lines of JavaScript involving bit operations.

We conduct two sets of experiments. For the first set we use `ab` [ab], which is considered the de-facto tool for benchmarking an Apache web server, over a Fast Ethernet (FE) network. We configure `ab` to issue 100 requests for the heavy, normal and light web pages, while the xJS module is enabled. Then, we perform the same experiments using the tests and with the xJS Apache module removed. We repeat all the above with the `ab` client running in a typical downstream DSL line (8 Mbps).

Figure 4.2 summarizes the results for the case of the `ab` tool connecting to the web server through a FE connection. The modified Apache imposes an overhead that ranges from a few (less than 6 ms and less than 2 ms for the normal and light test, respectively) to tens

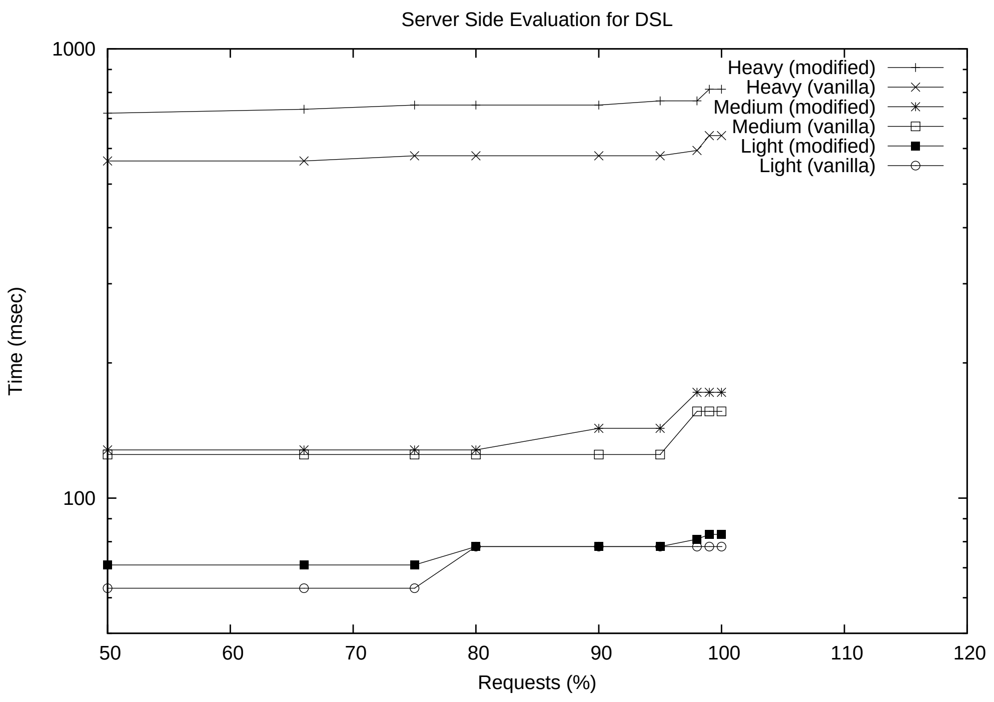


Figure 4.3: Server side evaluation when the Apache benchmark tool (`ab`) is requesting each web page through a DSL link. In the worst case (heavy) the server imposes a fixed delay of a few tens of milliseconds, like in the case of the Fast Ethernet setup (see Figure 4.2). However, this delay does not dominate the overall delivery time.

of milliseconds (about 60 ms) in the worst case (the heavy web page). While the results are quite promising for the majority of the tests, the processing time for the heavy page could be considered significant. In Figure 4.3 we present the same experiments over the DSL link. The overhead is still the same and it is negligible (less than a roundtrip in today's Internet) since now the delivery overhead dominates. This drives us to conclude that the Apache module imposes a fixed overhead of a few milliseconds per page, which is not the dominating overhead.

4.4.3 Client Overhead

Having examined the server-side overhead, we now measure the overhead imposed on the browser by xJS. We use the SunSpider test suite with 100 iterations, with every test executed 100 times. We use the `gettimeofday()` function to measure the execution time of the modified functions in each browser. Each implementation has two functions altered. The one that is responsible for handling code associated with events, such as `onclick`, `onload`, etc., and the

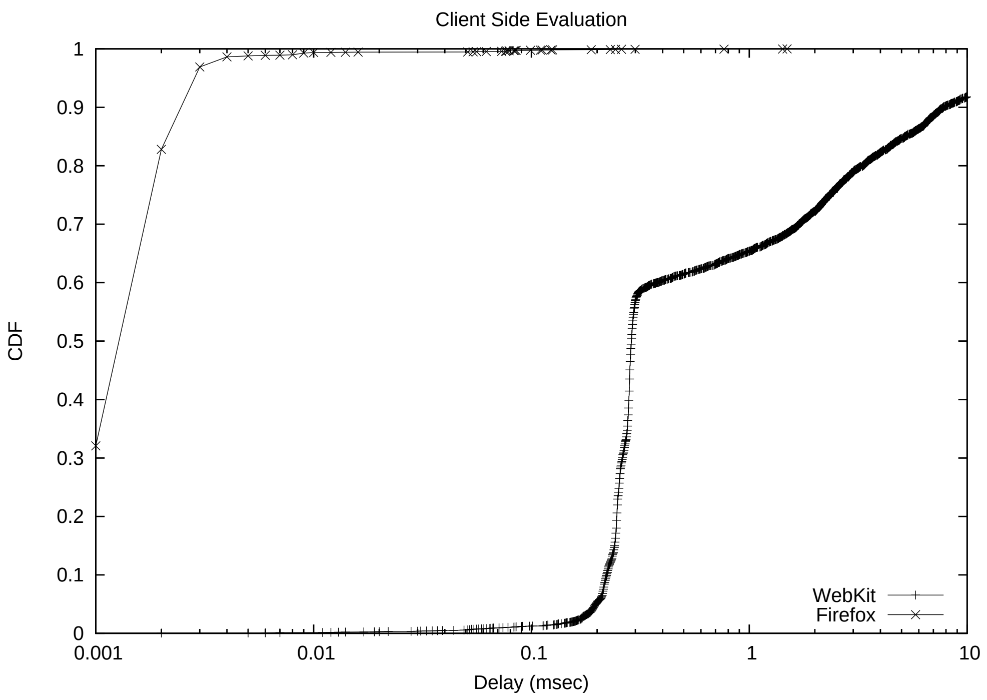


Figure 4.4: Cumulative distribution for the delay imposed by all modified function calls in the Firefox and WebKit implementation, respectively. As delay we assume the time needed for the modified function to complete minus the time needed for the unmodified one to complete. Notice that the majority of function calls imposes a delay of a few milliseconds.

one that is responsible for evaluating code blocks of JavaScript. The modifications of Firefox are substantially different. In Firefox we have modified internally the JavaScript `eval()` function which is recursive. These differences affect the experimental results in the following way. In WebKit we record fewer long calls in contrast with Firefox, in which we record many short calls.

In Figure 4.4 we present the cumulative distribution of the delays imposed by all modified recorded function calls for Firefox and WebKit during a run of the SunSpider suite for 100 iterations. As delay we define the time needed for the modified function to complete minus the time needed for the unmodified one to complete. Observe that the Firefox implementation seems to be the faster one. All delays are less than 1 millisecond. However, recall that Firefox is using a lot of short calls, compared to WebKit. Firefox needs about 500,000 calls for the 100 iterations of the complete test suite. In Figure 4.4 we plot the first 5,000 calls for Firefox (these calls correspond to one iteration only) of the complete set of about 500,000 calls, for

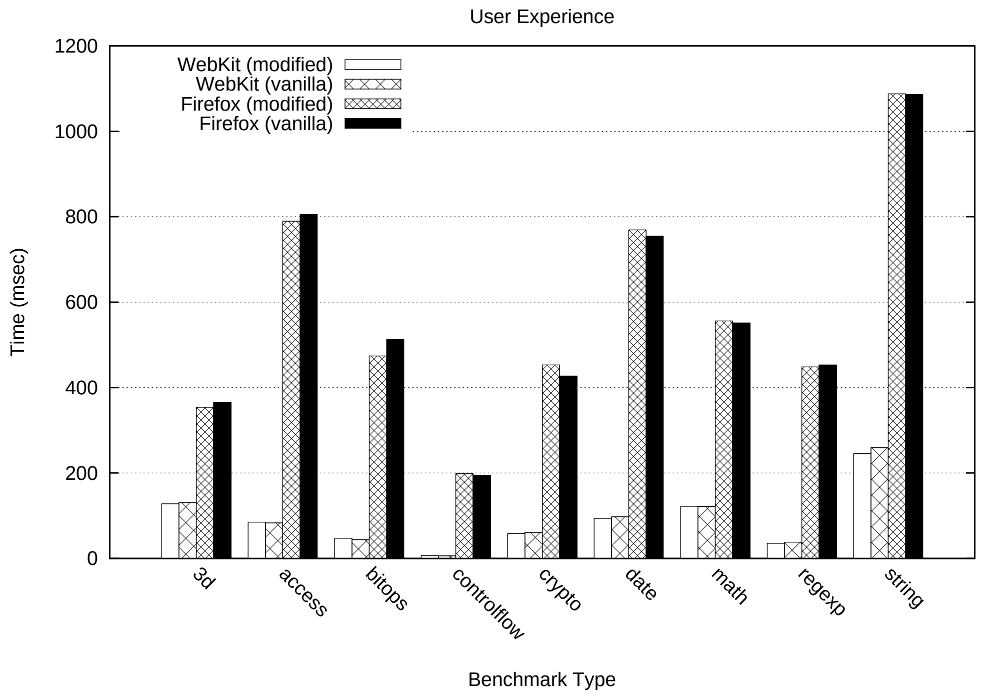


Figure 4.5: Results from the SunSpider test suite. Notice that for each modified browser the results are comparable with the results of its unmodified equivalent. That is, all de-isolated JavaScript executes as expected in both modified and unmodified browser.

visualization purposes and to facilitate comparison, and all 4,800 calls needed for WebKit to complete the test suite, respectively. The majority of WebKit's calls impose an overhead of less than one millisecond.

4.4.4 User Browsing Experience

We now identify whether user's browsing experience changes due to xJS. As user browsing experience we define the performance of the browser's JavaScript engine (i.e., running time), which would reflect the user-perceived rendering time (as far as the JavaScript content is concerned) for the page. We run the SunSpider suite *as-is* for 100 iterations with all three modified web browsers and with the equivalent unmodified ones and record the output of the benchmark. In Figure 4.5 we plot the results for all different categories of tests. Each category includes a few individual benchmark tests. As expected there is no difference between a modified and a non-modified web browser for both Firefox and WebKit. This result is reasonable, since after the de-isolation process the whole JavaScript source executes normally as it is in the case with a non compatible with the xJS framework web browser. Moreover, this experiment shows that xJS is not restrictive with legitimate web sites, since all the SunSpider suite (some thousands of JavaScript LoCs) run without any problem or side-effect.

4.5 Discussion

We now discuss potential limitations of our approach and offer possible workarounds.

JavaScript Obfuscation. Web pages served by xJS have all JavaScript encoded in Base64. Depending on the context this may be considered as a feature or not. For example, there are plenty of custom tools that obfuscate JavaScript on purpose. Such tools are used by certain web sites for protecting their JavaScript code and prevent visitors from copying the code. We should make clear that emitting all JavaScript encoded does not harden the development process, since all JavaScript manipulation takes place during serving time. While debugging, web developers may safely switch off xJS. Blueprint [TLV09] also emits parts of a web page in Base64.

eval() Semantics and Dynamic Code. As previously discussed (see Section 4.4), in order for xJS to cope with XSS attacks that are based on malicious injected data, the semantics of `eval()` must change. More precisely, our Firefox modifications alter the `eval()` function in the following way. Instead of simply evaluating a JavaScript content, the modified `eval()` function performs de-isolation before evaluation. This behavior can break web applications that are based on the generation of dynamic JavaScript code, which is executed using `eval()` at serving time. While this type of programming might be considered inefficient and error-prone, we suggest the following workaround. The JavaScript engine can be enhanced with an `xeval()` variant which does not perform any de-isolation before evaluation. The web programmer must explicitly call `xeval()` if this is the desired behavior. Still, there is no possibility for the attacker to evaluate her code (using `xeval()`), since the original call to `xeval()` must be already isolated.

Code Templates and Persistent XSS. Web developers frequently use templates in order to produce the final web pages. These templates are stored usually in a database and sometimes they include JavaScript. The database may also contain data produced by user inputs. In such cases, the code injection may take place *within* the database (persistent XSS). This may occur if trusted code and a user input containing malicious code are merged together before included in the final web page. This case is especially hard to track, since it involves the programmer's logic to a great extent. The challenge lies in that client-side code is hosted in another environment (the database) which is also vulnerable to code injections. xJS assumes that all trusted JavaScript is stored in files and not in a database. If the web developer wishes to store legitimate JavaScript in a database then she can place it in read-only tables. With these assumptions, xJS can cope with persistent XSS. Recall from Section 4.2 that xJS module is the first to run in the Apache module chain and, thus, all JavaScript isolation will take place before any content is fetched from databases or other external sources.

Chapter 5

Applying ISR Directly To JavaScript

5.1 Overview

RaJa applies randomization directly to JavaScript source. We modify a popular JavaScript engine, Mozilla SpiderMonkey [spi], to carefully modify all JavaScript identifiers and leave all JavaScript literals, expressions, reserved words and JavaScript specific constructs intact. We further augment the engine to recognize tokens identifying the existence of a third-party programming language. This is driven by two observations:

- JavaScript usually mixes with one and only one server-side scripting language, like PHP, with well defined starting and ending delimiters.
- Server-side scripting elements when are mixed-up with JavaScript source, they act as JavaScript identifiers or literals in the majority of the cases (see Figure 5.1).

To verify these speculations we deploy RaJa in four popular web applications. RaJa fails to randomize 9.5% of identified JavaScript in approximately half a million lines of code, which contain JavaScript, PHP and markup (HTML/XML). A manual inspection of the failed cases suggests that failures are due to coding idioms that can be grouped in *five* specific practices. Moreover, these coding practices can be substituted with alternative ones. For example, a

```
<script>
var message =
  <?php echo "Hello World<br/>"; ?>;
document.write(message);
</script>
```

Figure 5.1: PHP mixes with JavaScript. In the majority of the cases, the PHP expression acts as a JavaScript identifier or literal. In this example it acts as a string literal.

web application of approximately 150,000 lines of code, like WordPress, needs only a few code changes to be RaJa-enabled out of the box. RaJa is not just a randomization scheme, but rather a complete architecture for randomizing all JavaScript source of a web application, without the web programmer being aware of it. More precisely, RaJa is based on the following assumptions:

1. All JavaScript stored in files on the server's disk is considered trusted.

2. Only the JavaScript stored in files on the server's disk is considered trusted

The second assumption can be relaxed to account for JavaScript templates in databases, if read-only tables for this particular purpose are introduced. Based on these assumptions, RaJa can be enabled in a web application by simply placing the web application's code in a web server that runs the RaJa tools and libraries. We discuss all components of the RaJa architecture in Section 5.2.

5.2 Architecture

In this section we present in detail the RaJa architecture. We highlight all key components that compose the architecture and code modifications we have performed in existing software. RaJa builds on well-known and highly used open source projects like Mozilla and Apache. Finally, we give a short overview in a collection of tools we have specifically build for system administrators who want to utilize the RaJa framework.

5.2.1 Overview

RaJa is based on the idea of Instruction Set Randomization (ISR) to counter code injections in the web environment. XSS, the most popular code-injection attack in web applications, is usually carried out in JavaScript and, thus, RaJa aims on applying ISR to JavaScript. However, the basic corpus of the architecture can be used in a similar fashion for other client-side technologies (for example in Adobe Flash and Microsoft Silverlight applications).

In a nutshell, RaJa takes as input a web page and produces a new one with all JavaScript randomized. A simple example is shown in Figure 5.2. Notice that in the randomized web page all JavaScript variables (underlined in the Figure) are concatenated with the random token `0x78`. All other HTML elements and JavaScript reserved tokens (like `var` and `if`) as well as JavaScript literals (like `"Hello World"`, `"welcome"` and `true`) have been kept intact. The randomized web page can be rendered in a web browser that can de-randomize the JavaScript source using the random token. RaJa needs modifications both in the web server and the web client, as is the case of many XSS mitigation frameworks [JSH07, NSS09, GC09, APK+10].

In order to perform the randomization, RaJa needs to run as a pre-processor before any other server-side language (like PHP) takes place. RaJa assumes that only the JavaScript stored in files[1] of the server is trusted. Randomizing all trusted JavaScript ensures that any code injections will not be able to execute in a web browser that supports the framework.

A sample work-flow of a RaJa request-response communication is as follows. The RaJa-compliant web browser announces that it supports the framework using an `HTTP Accept`[2] header. The web server in turn opens all files needed for serving the requests and randomizes each one with a unique per-request key. Typically, a request involves several files that potentially host JavaScript, which are included in the final document through a server-side language. For example, PHP uses `require` and similar functions to paste the source of a document in the final web response. RaJa makes sure that all JavaScript involved is randomized. Finally, the web server attaches an `HTTP X-RAJA-KEY` header field which contains the

[1]This assumption can be augmented to support JavaScript stored in a database if we introduce read-only tables (see discussion in Section 5.5).

[2]For the definition of the `HTTP Accept` field, see: `http://www.w3.org/Protocols/HTTP/HTRQ_Headers.html#z3`

```
<!-- Original Document.  -->
<html>
<script>
var s = "Hello World!";
if (true)
  document.
    getElementByName("welcome").
    text = s;
</script>
<div id="welcome"></div>
</html>

<!-- Randomized Document.  -->
<html>
<script>
var s0x78 = "Hello World!";
if (true)
  document0x78.
    getElementByName0x78("welcome").
    text0x78 = s0x78;
</script>
<div id="welcome"></div>
</html>
```

Figure 5.2: A typical RaJa example.

randomization key. The RaJa-compliant web browser can then de-randomize and execute all trusted JavaScript. Any JavaScript source code that is not randomized can be detected and prevented for executing. The potential code injection is logged in a file. We now look into details on how we enable RaJa in the server and client side, respectively.

5.2.2 Server Side

In order to enable RaJa in a web server we use two basic components: (a) an Apache module (`mod_raja.so`), which operates as content generator and (b) a library interceptor which handles all `open()` calls issued by Apache. Although RaJa can be used with any server-side technology, for the purposes of this thesis we use PHP. Thus, we have configured the RaJa-enabled web server to use PHP as an output filter. For all experiments we use PHP acting as an output filter, but if someone prefers to use PHP as a module and not as a filter, two

Apache web servers can be used with the RaJa-enabled Apache acting as a proxy to the PHP-enabled one.

The RaJa Apache module handles initially all incoming requests for files having an extension of `.html`, `.js` and `.php`. This can be configured to support many other file types. For each request, it generates a random key and places it to a shared memory placeholder. It then opens the file in order to fulfill the request. The call to `open()` is intercepted using the `LD_PRELOAD` [ldp] functionality, available in most modern operating systems, by the RaJa randomizer, which acts as follows. It opens the file and tries to identify all possible JavaScript occurrences. That is, all code inside a `<script>` tag, as well as all code in HTML events such as `onclick`, `onload`, etc. For every JavaScript occurrence a parser, based on the Mozilla SpiderMonkey [spi] JavaScript engine is invoked to produce the randomized source. All code is randomized using the token which is retrieved from the shared memory placeholder. We analyze in more detail the internals of the SpiderMonkey-based parser below.

The randomized code is placed in a temporary file and the actual `libc_open()` is called with the pathname of the randomized source. Execution is transferred back to the Apache RaJa module. The module takes care for two things. First, it attaches the correct `Content-Length` header field, since the size of the initial file has possibly changed (due to the extra tokens attached to JavaScript source). Second, it attaches the `X-RAJA-KEY` header field to the HTTP response, which contains the token for the de-randomization process. The key is refreshed per request. All randomized code is contained in an internal memory buffer. This buffer is pushed to the next operating element in the Apache module chain. If the original request is for a PHP file, then the buffer will be pushed to the PHP output filter. It is possible that PHP will subsequently open several files while processing `require()` or similar functions. Each `open()` issued by the PHP filter is also intercepted by the RaJa randomizer and the procedure is repeated again until all PHP work has been completed. The size of the final response has possibly changed again, due to the PHP processing. PHP takes care for updating the `Content-Length` header field.

We present the control flow of the RaJa architecture in Figure 5.3 with all eight steps enumerated. We now proceed and present a step-by-step explanation of a RaJa-enabled

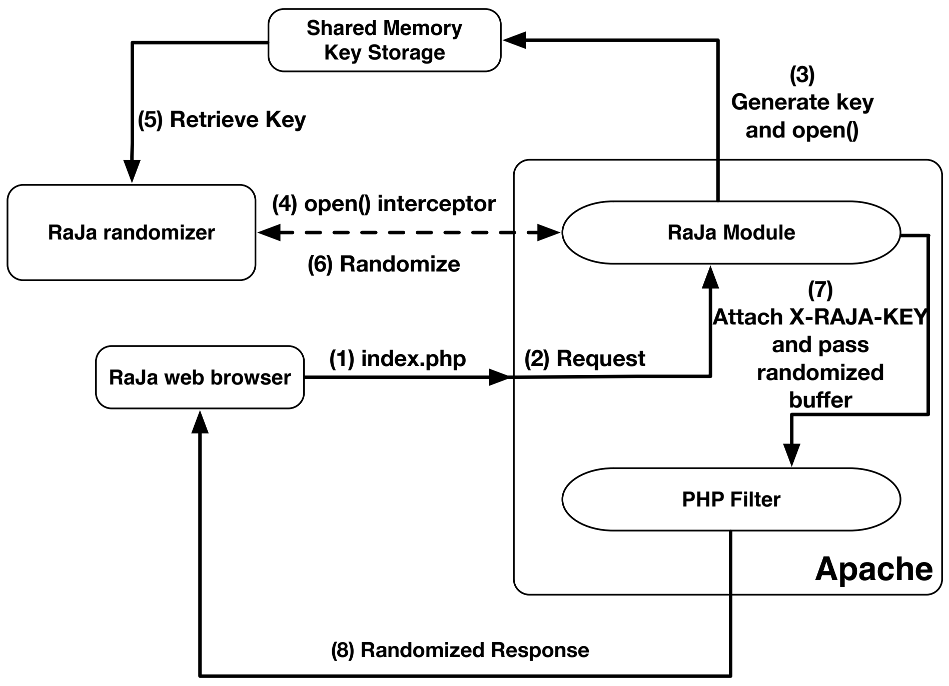


Figure 5.3: Schematic diagram of the RaJa architecture.

request-response communication. In Step (1) the RaJa-enabled web client requests `index.php` from a RaJa-enabled web server. In Step (2) the request is forwarded to the RaJa module which in turn in Step (3) generates a key, stores the key in a shared memory fragment and opens the file `index.php`. In Step (4) the RaJa randomizer intercepts `open()` and in Step (5) it retrieves the key from the shared memory fragment. In Step (6) `index.php` is opened, randomized, saved to the disk in a temporary file and the actual `libc_open()` is called with the pathname of the just created file. In Step (7) control is transferred to the RaJa module which adds the correct `Content-Length` and `X-RAJA-KEY` header fields. If the file is to be processed by PHP the buffer containing the randomized source is passed to the PHP filter. All `open()` calls issued from PHP will be further intercepted by the randomizer but we have omitted this in Figure 5.3 to make the graph more clear to the reader. Finally, in Step (8) the final document is served to the RaJa-enabled web browser.

5.2.3 Randomization

All JavaScript randomization is handled through a custom parser based on the SpiderMonkey [spi] JavaScript engine. The RaJa parser takes as input JavaScript source code and it produces an output with all code randomized. For an example refer to Figure 5.2. The original SpiderMonkey interpreter parses and evaluates JavaScript code without executing it. Instead, all source is printed randomized with all JavaScript identifiers concatenated with a random token. Special care must be taken for various cases. We enumerate a few of them.

1. Literals. All literals, like strings and numbers, are parsed and directly pasted in the output in their original form.

2. Keywords. All keywords, like `if`, `while`, etc., are parsed and directly pasted in the output in their original form.

3. HTML comments. The original SpiderMonkey removes all comments before evaluation. The RaJa parser pastes all HTML comments in their original form.

4. Language mixing. Typically a web page has a mixture of languages such as HTML, PHP, XML and JavaScript. The RaJa parser can be configured to handle extra delimiters as it does with HTML comments and thus identify other languages, such as PHP, which heavily intermix with JavaScript. We further refer to these languages as *alien languages*.

We augment the RaJa parser to treat occurrences of alien languages inside JavaScript according to the following rules.

- *Rule 1.* An alien language occurrence is treated as a JavaScript identifier if it occurs inside a JavaScript expression.

- *Rule 2.* An alien language occurrence is treated as a JavaScript comment and is left intact if *Rule 1* does not apply.

We conclude to these basic two rules after investigating four popular and large, in terms of lines of code (LoCs), web applications. By manually checking how PHP is mixing with

```
 1 <!-- Original Source.  -->
 2 <?php if (user_exists($user)) { ?>
 3  var message = <?php echo "Welcome" ?>;
 4 <?php } else { ?>
 5  var message = "Sign-Up Needed.";
 6 <?php } ?>
 7
 8 <!-- Randomized Source.  -->
 9 <?php if (user_exists($user)) { }?>
10  var message0x78 =
11   <?php echo "Welcome" ?>;
12 <?php } else { ?>
13  var message0x78 = "Sign-Up Needed.";
14 <?php } ?>
```

Figure 5.4: Code mixing of JavaScript with alien languages. In this example PHP is used as an alien language example. In line 3, Rule 1 is applied, while in lines 2, 4 and 6, Rule 2 is applied.

JavaScript, we observed that in the majority of the cases PHP serves as an identifier or literal inside a JavaScript expression (see line 3 in Figure 5.4). For a short example of how these two rules are applied refer to Figure 5.4.

5.2.4 De-randomization

The de-randomization process takes place inside a RaJa-compliant browser. In our case this is Firefox with an altered SpiderMonkey engine. The modified JavaScript interpreter is initialized with the random token, taken from the `X-RAJA-KEY` header field. During the parsing phase it checks every identifier it scans for the random token. If the token is found, the internal structure of the interpreter that holds the particular identifier is changed so as to hold the identifier de-randomized (i.e., the random token is removed). If the token is not found, the execution of the script is suspended and its source is logged as suspicious.

We take special care in order to assist in coding practices that involve dynamic code generation and explicit execution using the JavaScript's built-in function `eval()`. Each time a correctly randomized `eval()` is invoked in a script, the argument of `eval()` is not de-randomized. Notice that this is consistent with the security guarantees of the RaJa framework, since the `eval()` function is randomized in the first place and cannot be called explicitly by a malicious script unless the random token is somehow revealed. However, this approach is vulnerable to injections through malicious data that can be injected in careless use of `eval()`. For this case the RaJa framework can be augmented with tainting [Sek09, VNJ+07, NLC07, NtGG+05].

Self-Correctness. In order to prove that the RaJa parser does not produce invalid JavaScript source we use the built-in test-suite of the SpiderMonkey engine. We first run the test-suite with the original SpiderMonkey interpreter and record all failures. These failures are produced by JavaScript features which are now considered obsolete. We subsequently randomize all tests, remove all E4X [ECM04] tests because we do not support this dialect, re-run the test-suite with the `raja-eval` (a tool capable in executing randomized source) and record all failures. The failures are exactly the same. Thus, the modified SpiderMonkey behaves exactly as the original one in terms of JavaScript semantics.

5.2.5 Configuration and Administrator Tools

The RaJa framework can be configured with various dynamic options in order to be deployed in a web site. For example, it can be configured with the delimiters identifying an alien language. In Figure 5.5, we depict a sample configuration. It also includes a command-line tool-chain that can assist a web programmer or system administrator. These are the followings:

1. `raja`: Takes as input a token and a JavaScript source through standard input or a file and produces its randomized equivalent.

2. `raja-eval`: Takes as input a token and a randomized Javascript source code through standard input or a file and evaluates it.

3. `raja-enforce`: Takes as input a token and a web page and identifies how many scripts can be randomized and how many cannot. The latter works as follows. The first processing unit of the tool takes as input the web page and attempts to identify all possible JavaScript source code occurrences. It first removes PHP and HTML comments and

```
# RaJa Configuration.
# Path to RaJa randomizer.
randomizer = /home/raja/framework/raja
# Files handled.
ext = php, html, js
# Meta-tokens.
extra-tokens = <?php|?>
```

Figure 5.5: A typical RaJa configuration file.

then searches for JavaScript. That is, all code inside a `<script>` tag, as well as all code in HTML events such as `onclick`, `onload`, etc. Notice that the identified code may contain PHP elements. The mixed JavaScript source code is stored for further processing. The second processing unit is the modified JavaScript parser. For every stored source code from the previous phase, the parser is invoked to produce the randomized output. In case the tool fails to randomize the script it outputs a syntax error.

5.3 Case Studies

In this section we test RaJa with popular web applications. We present our experiences from deploying the framework with existing source code, which is composed by multiple flavors of server-side and client-side code. We, also, highlight various coding idioms and practices we found, while trying to enable RaJa in real-world web applications.

5.3.1 WordPress

WordPress is a popular blog engine based on PHP and MySQL. The web application is composed by approximately 150,000 lines of code (version 2.9). The source corpus includes PHP, HTML, XML, SQL and JavaScript code, mixed up in various ways. WordPress had many security vulnerabilities in the past and thus it is considered ideal for testing RaJa.

We install WordPress in an Apache that supports RaJa. We use our tools introduced in Section 5.2 to scan all the source code of WordPress and spot JavaScript snippets. We attempt to randomize all 187 identified scripts. The RaJa randomizer manages to successfully randomize 169 scripts and fails to randomize 18. The failed tests indicate that the randomizer cannot process the scripts up to the end due to code interference between JavaScript and PHP. The successfully passed scripts indicate that the randomizer succeeded on them. It is questionable if the rest of the PHP code has been modified wrongly due to the randomizer. We further proceed and check each file using PHP's lint mode (i.e. invoke the PHP interpreter using the "`-l`" option) to check the randomized files syntactically. All files succeed on passing the PHP syntax check. This means that the randomization process does not introduce errors in the PHP code.

We manually analyze all failed tests and categorize the problems that drive the RaJa randomizer to failure. We conclude that there are five general cases where code-mixing between PHP and JavaScript results in broken randomized source:

- *Case 1.* Partial injection of JavaScript source using the PHP built-in function `echo()`. Lines 2-6 in Figure 5.6.
- *Case 2.* String concatenation. Lines 9-10 in Figure 5.6.
- *Case 3.* Partial JavaScript code generation by PHP scripting blocks. Lines 13-17 in Figure 5.6.
- *Case 4.* JavaScript code generation by using frameworks' meta languages. Lines 20-23 in Figure 5.6.
- *Case 5.* Markup injections. Lines 26-30 in Figure 5.6.

We depict examples of all five cases that force RaJa to produce faulty randomized source code in Figure 5.6. All examples are from the source of WordPress, except for Case 3, where we use an example from phpBB and for Case 5, where we use an example from phpMyAdmin. Some examples are slightly altered for better presentation. We now proceed and discuss each case in detail and provide a workaround where possible.

Case 1. RaJa cannot handle partial fragments of JavaScript, since the source must be first fully parsed and then randomized. It's hard for the parser to isolate portions of JavaScript injected using the PHP built-in `echo()` function and then re-assemble them in a snippet that can be successfully parsed. The example of this case is depicted in Figure 5.6 (lines 2-6) and it can be substituted with a different coding practice suggested in Figure 5.7 (lines 2-6).

Case 2. RaJa cannot handle complex string concatenation of mixed JavaScript and PHP variables, since both languages support all quotation flavors in string literals. Thus, string quoting in PHP interferes with JavaScript quoting and it is hard to isolate one from the other. The example of this case is depicted in Figure 5.6 (lines 9-10) and it can be substituted with a different coding practice suggested in Figure 5.7 (lines 9-11).

Case 3. RaJa treats all alien language occurrences as an identifier according to Rule 1 (see Section 5.2). There are some cases where PHP code is injected in a location that cannot be handled correctly as a JavaScript identifier (recall Rule 1 from Section 5.2). Consider the case in line 15 of Figure 5.6. The curly brackets surrounding the PHP code denote an object inside. Thus, the JavaScript interpreter does not expect to parse a common identifier. The example of this case is depicted in Figure 5.6 (lines 13-17) and it can be substituted with a different coding practice suggested in Figure 5.7 (lines 14-19).

Case 4. RaJa cannot randomize JavaScript when it is mixed with meta language elements. The JavaScript source cannot be parsed if the meta language code is removed, since the removal leaves two subsequent JavaScript identifiers, which in turn produce a syntax error. The example of this case is depicted in Figure 5.6 (lines 20-23) and it can be overcome either by extending the parser to identify some meta language elements and ignore them or by giving a different coding practice suggested in Figure 5.7 (lines 22-26).

Case 5. RaJa cannot randomize JavaScript when it is mixed with HTML special characters. These HTML characters are expanded by a PHP filter before the script reaches the web browser. Before the expansion the JavaScript expression is invalid. For example, the most frequently occurring case is when the sequence of HTML entities `&amp;&amp;` is expanded to `&&`, which is the logical `AND` in a JavaScript expression. We are not aware of the goal of the web programmers that use this coding tactic. The example of this case is depicted in Figure 5.6 (lines 26-30) and it can be overcome by extending the parser to ignore all HTML special characters.

All workarounds suggested are meant for assisting RaJa to handle complex code mixing

Application	LoCs	Scripts	Passed	Failed
WordPress	143,791	187	169	18
phpBB	213,681	539	512	27
phpMyAdmin	178,447	263	215	48
Drupal	44,780	8	6	2
Total	580,699	997	902	95

Table 5.1: Summary of scripts that RaJa can successfully randomize in four real-world web applications.

between JavaScript and PHP. Web programmers may dislike the coding idioms we suggest and it is not our intention to enforce coding practices. However, we believe that programming aesthetics may be overlooked in favor of security.

We further proceed and alter all faulty scripts according to the workarounds we suggest above. The result is a full functional and RaJa-enabled WordPress. We manage to create and administrate a sample blog using a RaJa-enabled web browser. We present an example from a randomized script from the WordPress distribution in Figure 5.8.

5.3.2 phpBB, phpMyAdmin and Drupal

We perform similar studies for other three large and popular web applications, namely phpBB, phpMyAdmin and Drupal. In Table 5.1 we present a summary of our complete study. Specifically, we list all JavaScript snippets identified, the number of them that we can successfully randomize, as well as all scripts that fail in randomization.

All failed scripts are covered by the five cases we have already presented. It is worth mentioning that phpBB uses internally a meta-language for better code structuring (see example of Case 3 in Figure 5.6). RaJa can effectively identify and randomize most of phpBB scripts even if the language mixing is triple (PHP, JavaScript and phpBB meta-language). In Table 5.2 we present all scripts in all four web applications that failed due to one of the five cases mentioned above.

Web Application	Scripts	C1	C2	C3	C4	C5
WordPress	18	3	12	3	0	0
phpBB	27	1	0	0	26	0
phpMyAdmin	48	0	43	2	0	3
Drupal	2	0	2	0	0	0
Total	95	4	57	5	26	3

Table 5.2: Categorization of all mixed scripts in all four web applications.

5.3.3 Summary

In this section we performed an extensive study of four real-world web applications. We manually spotted scripts that cannot be randomized due to language mixing. However, RaJa can locate and randomize the majority of the JavaScript in all four web applications. More precisely, RaJa fails to randomize 9.5% of identified JavaScript in approximately half a million lines of code, mixed up with JavaScript, PHP and markup. Plan of our future work is to experiment with a more tolerant JavaScript parser that can recover from syntax errors introduced due to heavily language mixing.

5.4 Evaluation

In this section we evaluate RaJa. We measure the overhead RaJa imposes on the server and client side and we test the framework with real-world attacks from a well-known XSS repository.

Server-Side Overhead. The most crucial part of RaJa performance is the server-side part. All web pages are examined for JavaScript source code and if an occurrence is found the RaJa randomizer performs a full-JavaScript parsing session. This process is repeated for every request (see Section 5.5 about caching issues). This is vital for the security guarantees of the framework, since the randomization key has to be refreshed in every request. Otherwise, an XSS exploit can perform a request to discover the key and then launch the actual attack.

In order to measure the overhead imposed on the server-side part, we request a set of web pages that embed a significant amount of JavaScript using the Apache Benchmark (`ab`) [ab]. The sample of web pages is collected from the SunSpider [sun] suite, which constitutes a

```
/* Case 1.  */
echo "<script type='text/javascript'>\n";
echo "/* <![CDATA[ */\n";
echo "document.write(...);";
echo "/* ]]> */\n";
echo "</script>\n";

/* Case 2.  */
$actions['quickedit'] =
 'onclick="Reply.open(\'' . $post->ID . '\',\'edit\');"'

/* Case 3.  */
tinyMCEPreInit = {
 ...
 mceInit : {<?php echo $mce_options; ?>},
 ...
};

/* Case 4.  */
var RecaptchaOptions = {
 lang : '{L_RECAPTCHA_LANG}',
 index : <!-- IF $C-->{$C}<!-- ELSE -->10<!-- ENDIF -->
};

/* Case 5.  */
onsubmit=
  "return
  (emptyFormElements(this, 'table')
  &amp;&amp; checkFormElementInRange(
  this, 'num_fields', ...);"
```

Figure 5.6: Categorization of all cases that result in faulty randomization due to interference between JavaScript and PHP.

```
/* Case 1.  */
<script type='text/javascript'>
/* <![CDATA[ */
document.write(...);
/* ]]> */
</script>

/* Case 2.  */
$content = '\'' . $post . '\',\'edit\'';
$actions['quickedit'] =
 'onclick="commentReply.open('.$content.');"';

/* Case 3.  */
<?php $mce_options_s = "{" . $mce_options . "}"; ?>
tinyMCEPreInit = {
 ...
 mceInit : <?php echo $mce_options_s; ?>,
 ...
};

/* Case 4.  */
<!-- IF $CAPTCHA -->
var RecaptchaOptions = { ... };
<!-- ELSE -->
var RecaptchaOptions = { ... };
<!-- ENDIF -->
```

Figure 5.7: Suggested workarounds for all cases that result in faulty randomization due to interference between JavaScript and PHP.

```
<script type="text/javascript">
 <?php  if ( $user_login || $interim_login ) { ?>
 setTimeout_0x123_( function(){ try{
 d_0x123_ = document_0x123_.getElementById_0x123_('pass');
 d_0x123_.value_0x123_ = '';
 d_0x123_.focus_0x123_();
 } catch(e_0x123_){}
 }, 200);
 <?php  } else { ?>
 try{
        document_0x123_.getElementById_0x123_('login').
                        focus_0x123_();
  }catch(e_0x123_){}
 <?php  } ?>
 </script>
```

Figure 5.8: Example of randomized source code from WordPress (`wp-login.php`).

collection of JavaScript benchmarks for measuring the performance of JavaScript engines. The suite is composed by nine different groups of programs that perform various complex operations. We manually select three JavaScript tests from the SunSpider suite. The *heavy* test involves string operations with many lines of JavaScript. This is probably the most processing-intensive test in the whole suite, composed by many lines of code. The *normal* test includes a typical amount of source code like most other tests that are part of the suite. Finally, the *light* test includes only a few lines of JavaScript involving bit operations.

We conduct two sets of experiments. For the first set we use `ab` over a Fast Ethernet (FE) network. We configure `ab` to issue 1000 requests for the heavy, normal and light web pages to stress a RaJa-enabled server. Then, we perform the same experiments using a vanilla Apache. We repeat all the above with the `ab` client running in a typical downstream DSL line.

Figure 5.9 summarizes the results for the case of the `ab` tool connecting to the web server through a FE connection. The RaJa server imposes an overhead that ranges from a few tens of milliseconds to about one hundred of milliseconds in the worst case (the heavy web page). While the results are quite promising for the majority of the tests, the processing time for the heavy page could be considered significant. However, in Figure 5.10 we present the same experiments over the DSL link. The overhead is still the same and it is negligible (less than

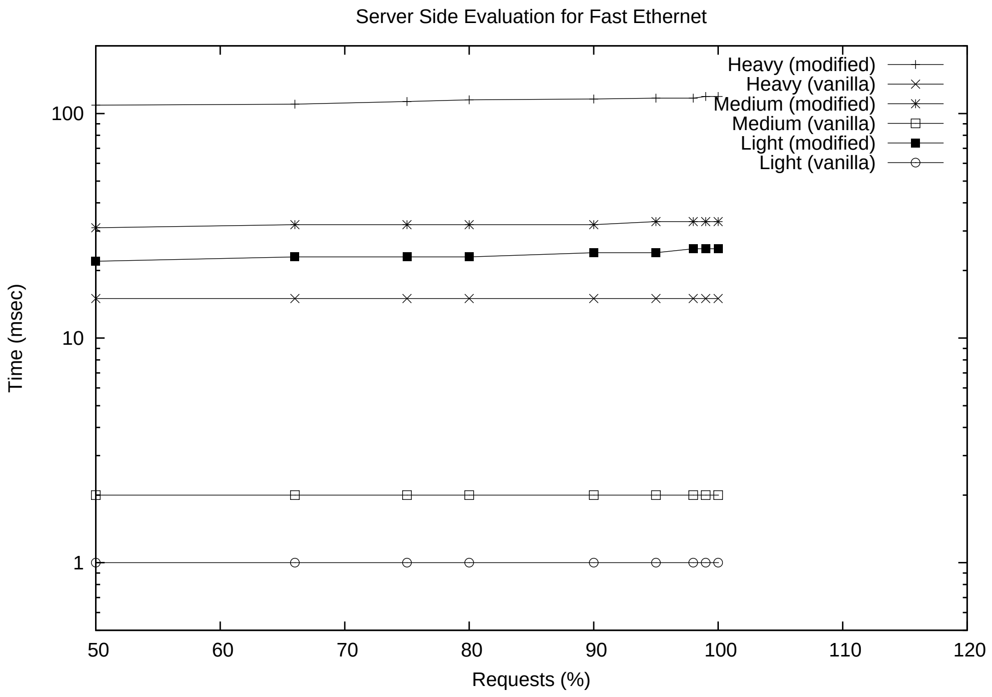


Figure 5.9: Server side evaluation when the Apache benchmark tool (`ab`) is requesting each web page through a Fast Ethernet link. In the worst case (heavy) the server imposes delay of about 100 of milliseconds, while in the normal case the delay is only a few milliseconds.

a roundtrip in today's Internet) since now the delivery overhead dominates. This drives us to conclude that RaJa imposes a fixed overhead of a few tens of milliseconds per page, which is not the dominating overhead.

Client-Side Overhead. The additional overhead in the client-side originates from the fact that a RaJa compliant web browser embeds a modified SpiderMonkey engine, which derandomizes each JavaScript identifier while parsing. We expect this overhead to be relatively small compared to the authentic SpiderMonkey engine, since it involves a string comparison, relatively to the size of the token, for every identifier parsed.

In order to evaluate the client-side overhead we use the SunSpider [sun] suite, which includes a collection of JavaScript benchmarks. These tests are meant to stress the JavaScript engine of a web browser. The benchmarking is carried-out using JavaScript which in the RaJa case is also randomized and cannot be accounted. However, the code involved for accounting the benchmarks is significantly lesser than the code of the tests and thus the results slightly deviate from the real overhead. We plot the benchmark results for all seven different families

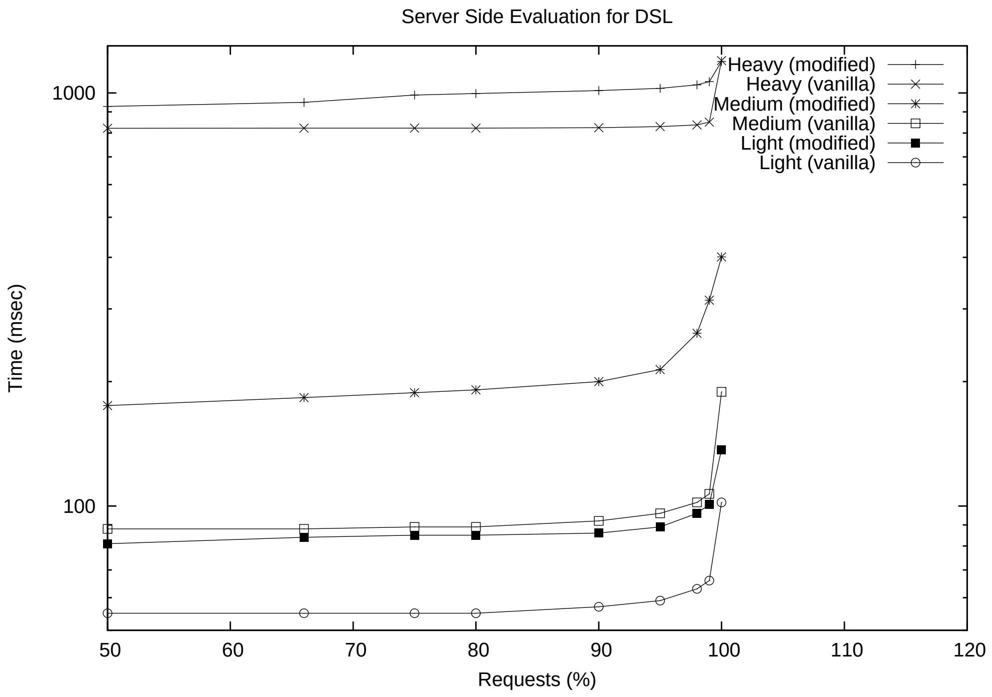


Figure 5.10: Server side evaluation when the Apache benchmark tool (`ab`) is requesting each web page through a DSL link. In the worst case (heavy) the server imposes a fixed delay of a few tens of milliseconds, like in the case of the Fast Ethernet setup (see Figure 5.9). However, this delay does not dominate the overall delivery time.

of tests in Figure 5.11. As expected, the RaJa-enabled web browser is slightly slower than the original Firefox.

Attack Coverage. RaJa is designed to detect and prevent all XSS *reflection* attacks. We use the repository hosted by XXSed.com [FP] which includes a few thousands of XSS vulnerable web pages. This repository has been also used for evaluation in other papers [NSS09, APK$^+$10]. The majority of these exploits refer to XSS reflection attacks, excluding some `<iframe>` injections. We evaluate RaJa, with XSS reflection attacks only, against all web sites that have not fixed the vulnerability since the exploit's publication, 1,401 in total. More precisely, we use the exact methodology presented in [APK$^+$10]. As expected all 1,401 exploits are successfully detected and prevented from executing.

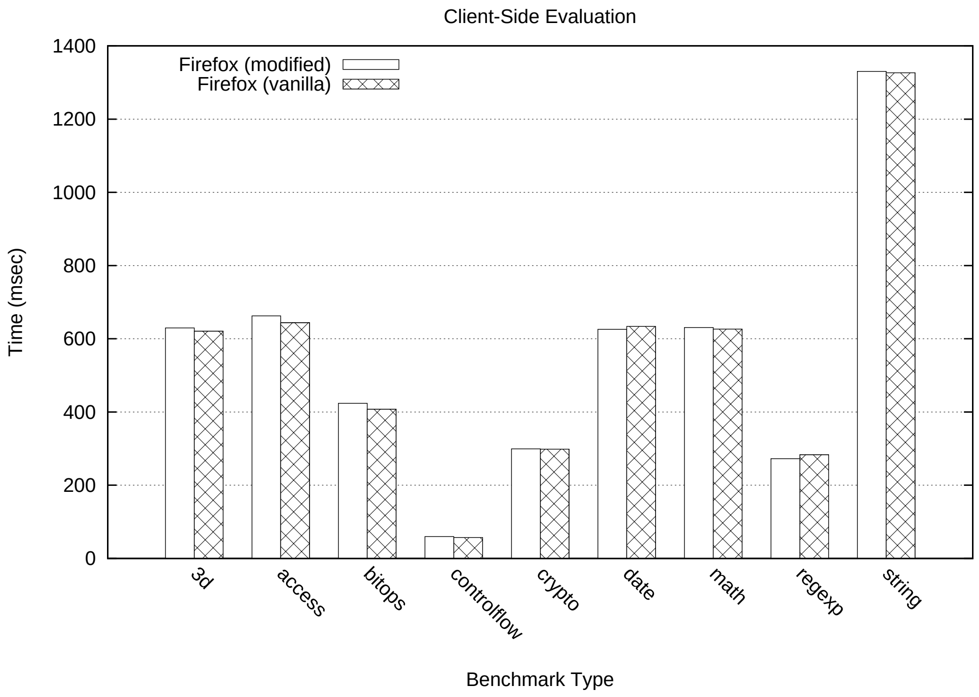


Figure 5.11: Client-side evaluation for a RaJa-enabled web browser using the SunSpider benchmarks. A RaJa-enabled web browser is slightly slower compared to Firefox.

5.5 Discussion

We now discuss some important aspects that we have not highlighted, but they are crucial for deploying RaJa.

JavaScript Templates in Databases. RaJa assumes that all legitimate JavaScript is stored in the web application's files. We consider that an attacker has not compromised the web server, since then an XSS attack is redundant. We also assume that incoming data from the network is not placed in files but in a database, since using the filesystem as a data storage for a web application is considered rather obsolete. However, there is the possibility that a web application stores legitimate JavaScript code in a database. Since, RaJa randomizes only JavaScript stored in files, code fetched from a database is considered as a code injection. We are not aware of any real web application that stores legitimate JavaScript in databases. However, it is worth mentioning that RaJa can potentially co-operate with such a scheme provided that the tables storing the legitimate code have been marked as read-only and thus cannot contain both legitimate and malicious code.

Caching. RaJa refreshes the token which serves as the randomization key in every request. This is crucial for the security guarantees of the framework. If the key is not updated in every request then an attacker can potentially launch an XSS attack in two phases. In the first phase the attacker reveals the key and in the second phase performs the injection. This strategy can break if a caching scheme is applied. For web pages that involve partial caching of page fragments, this strategy can even produce non-functional web pages. Cached fragments will have different randomization key from non-cached ones and the browser will not be able to de-randomize the complete JavaScript source correctly. However, JavaScript is usually associated with dynamic pages, where caching is hard to be applied effectively. In addition, when caching is applicable, web servers and browsers tend to cache graphic and video objects, which are usually larger in size than the JavaScript source code of a web application.

Chapter 6

xHunter

6.1 Introduction

Although XSS targets web applications, the popularity of the exploitation method along with the prevalence of the web has made XSS a dominant threat in computer systems [Sym, SAN]. Unfortunately, such a statement can be officially supported and quantified only by large IT industry members involved in computer security. Undoubtedly, voluntary efforts from individuals exist, in order to keep track of the attack landscape of XSS. For example, XSSed.com [FP] maintains a repository of XSS exploits. The amount of stored incidents in XSSed.com is an evidence that web sites are continuously threaten from XSS attacks. Although this particular XSS repository is invaluable to the research community, it can hardly assist in quantifying the real problem.

We argue that the academic and research community lacks the necessary tools for performing measurement studies and quantifying the threat constituted by XSS attacks. More precisely, we would like to have the tools for answering questions like the followings:

1. How often are web sites targeted with XSS attacks? Is XSS a frequent phenomenon in every-day web traffic?

2. Which web sites are the targets?

3. Are there any orchestrated XSS campaigns in world-wide scale?

4. How do *real* XSS exploits look like? XSSed.com maintains a large collection of vulnerable to XSS web sites along with their exploitation. However, many of the URLs listed contain proof-of-concept exploit code, like the use of the JavaScript `eval()` function to display a message, and not code that can cause harm.

To this end, we propose xHunter, a tool that can passively monitor the network for identifying suspicious URLs. xHunter does not aim on providing any defenses against XSS, but rather collect statistics about them. The fundamental assumption behind xHunter is that XSS attacks based on JavaScript code injection (which is the most frequently encountered case of XSS) are carried out through URLs that contain a part that can produce a JavaScript-valid syntax tree. Unfortunately, JavaScript has a very relaxed syntax and is very context independent. Even simple expressions that are usually encountered in URLs can produce a valid JavaScript syntax tree. In addition, web applications use custom encoding schemes making URL parsing hard. In this chapter, we present all challenges we encountered while designing xHunter along with a prototype of the tool.

Organization. This chapter is organized as follows. We present the architecture of xHunter in Section 6.2. We analyze in detail all major challenges xHunter has to deal with in Section 6.3. In Section 6.4 we present a preliminary evaluation of xHunter using a sample of about 11,000 malicious URLs collected from XSSed.com and a sample of 1,000 URLs collected from a monitoring point in an educational organization with about 1,000 users. In Chapter 7 we use xHunter to analyze the entire repository of XSSed.com.

6.2 xHunter Architecture

In this section we present the basic architecture of xHunter. We first provide a short background of XSS attacks and then we phrase the two basic assumptions that are fundamental for the tool's design. We finally present the work flow of an xHunter run using a real example taken from XSSed.com.

6.2.1 Overview

There are basically two large categories of XSS attacks: (a) reflected and (b) stored. During a reflected XSS attack the injected code is placed in a URL. Once the user clicks on the malicious URL the injected code executes. On the other hand, during a stored XSS attack, the adversary injects the malicious payload in some form of storage utilized by the vulnerable web application. For example, consider a web application that handles a blog engine and stores all data associated with the blog in a database. An attacker can post an article which encapsulates the malicious code. This code is rendered in the user's browser when the user's browser renders the blog article. Notice, that even in the case of stored XSS, the attack has been injected using a URL, since this is the only way the attacker can communicate with the web application.[1]

Assumption 1. The first fundamental assumption for xHunter is that XSS attacks are carried out through the transmission of URLs that contain the malicious code. The malicious code during an XSS attack is usually expressed in JavaScript. There are plenty of other methods for exploitation, like injections of an `iframe`, redirection or leveraging of other client-side technologies such as Flash, injections in file uploads [BCS09] or Phishing [DTH06]. xHunter focuses only in cases where the web attack is triggered via a URL that contains JavaScript. These cases constitute a significant fraction of XSS attacks.

Assumption 2. The second fundamental assumption for xHunter is that an XSS attack is mounted using a URL which contains a part that can produce a valid JavaScript parse tree.

Based on the two assumptions above, xHunter works roughly as follows. It takes as an input a URL. It scans the input for parts that can produce a valid JavaScript parse tree. The tool embeds the JavaScript engine of Mozilla SpiderMonkey [spi] for generating parse trees. A URL is considered suspicious if a part produces a JavaScript parse tree with a certain *depth*.

For example, consider the JavaScript snippet in Figure 6.1. xHunter can process the source and produce the syntax tree depicted in Figure 6.1. Notice the indentation of the parse tree. It is slightly different than the actual parse tree a JavaScript parser can export. The reason is that xHunter assigns a different *weight* to different parse nodes (i.e. tokens the parser

[1]XCS attacks that are mounted through another non-web channel are out the scope of this work [BBB09].

```
if (user_logged()) {
  alert(document.cookie);
}

LC:
 IF:
  LP:
   NAME:
  LC:
   SEMI:
    LP:
     NAME:
    DOT:
     NAME:
```

Figure 6.1: Parse tree of a JavaScript snippet as produced by xHunter.

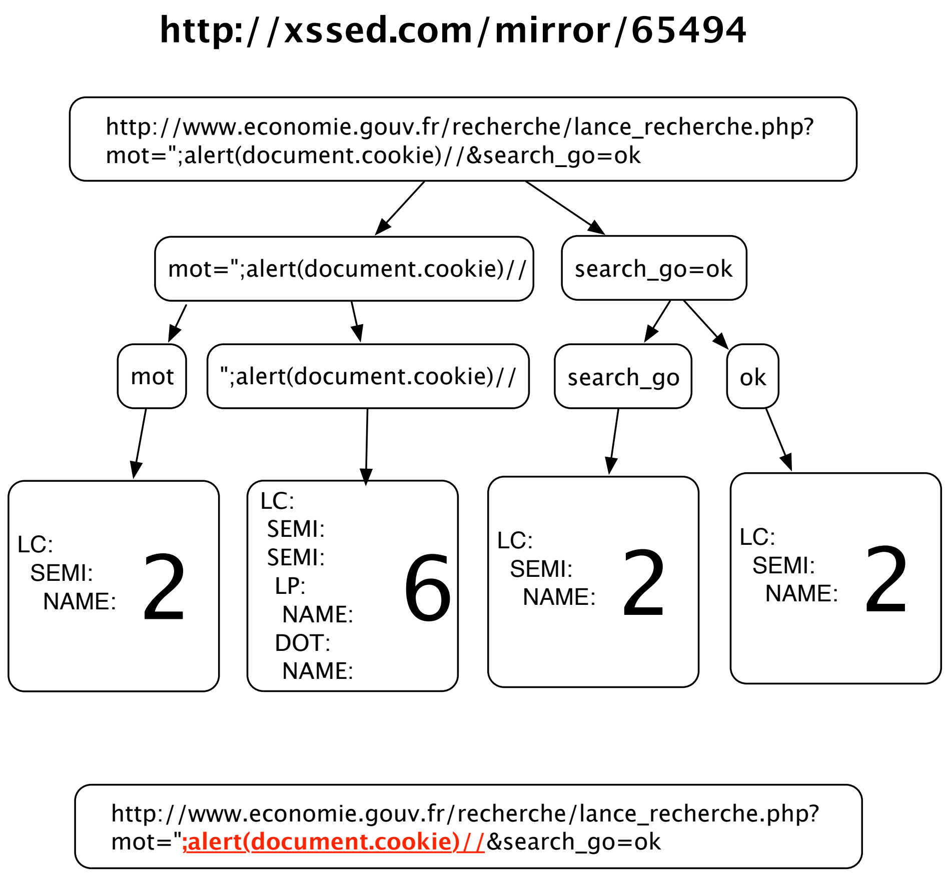
consumes). This weight is depicted in the figure as indentation level. We further analyze this later in Section 6.3. Notice also that xHunter does not evaluate the actual code. It is not aware of neither the DOM [LHLHW+04] structure nor the code of the web application the malicious code is trying to exploit. xHunter only checks if part of an input can produce a valid JavaScript syntax tree.

6.2.2 Operation

We now proceed and present a hypothetical xHunter run. xHunter takes as input a URL. First, the URL is HTML-escaped (for occurrences of HTML encoded entities like `&apos;`) and is URL-decoded. Then, the part of the URL which follows the "?" character, usually named as the *query string*, is isolated from the rest of the URL. The query string contains all parameters which take part in an HTTP GET request. [2] All parameters are separated using the "&" character. Each parameter is in the form `key = value`. For each pair, xHunter tries to parse both the key and the value (if they both exist). Apparently, there are XSS attacks that host the exploit code in the key and not in the value, as it is more common [xssa].

There are two possible outcomes when xHunter tries to parse a particular field. The field does not parse or the field produces a syntax tree with a certain depth. If the depth

[2] A similar approach is used for POST requests. In this case the query parameters are part of the HTTP request.



http://xssed.com/mirror/65494
http://www.economie.gouv.fr/recherche/lance_recherche.php?mot=";alert(document.cookie)//&search_go=ok
mot=";alert(document.cookie)//
search_go=ok
mot
";alert(document.cookie)//
search_go
ok
LC: SEMI: NAME: 2
LC: SEMI: SEMI: LP: NAME: DOT: NAME: 6
LC: SEMI: NAME: 2
LC: SEMI: NAME: 2
http://www.economie.gouv.fr/recherche/lance_recherche.php?mot=";alert(document.cookie)//&search_go=ok

Figure 6.2: Example operation of a hypothetical xHunter run.

exceeds a certain value, the URL is considered suspicious. The depth of the syntax tree depends on the tokens that compose the field's text. If there is JavaScript code in the field then the syntax tree is expected to have a depth of a high value. Notice, for example, in Figure 6.1 that the code responsible for the `alert()` code contains six different tokens (lines 9-14). Unfortunately, even simple text can produce a valid JavaScript syntax tree. Hopefully, real and possibly functional JavaScript code has higher probability to include certain tokens. Thus, each token is assigned with a weight and the overall depth is computed using each node's weight. We further discuss this in Section 6.3.

A theoretical xHunter run is depicted in Figure 6.2. The running example is a URL from XSSed.com that contains XSS attack code. Notice, how the query string and then the parameters are isolated from the original URL. Observe, also, that every field, in this particular case, produces a valid JavaScript syntax tree. Nevertheless, the attack code produces the syntax tree with the higher depth (6). Notice the final URL where the attack code has been underlined. The first character (double quotes) is not highlighted, although it is part of the field that contains the attack. This character has been omitted, since it produces a syntax error if included. We further discuss how we deal with partial JavaScript expressions in Section 6.3.

There are plenty of cases xHunter, as presented here, fails to handle. First, web applications support URLs that do not fully conform to the specification [BLMM94]. Second, JavaScript has a relaxed syntax and thus simple text can produce a valid syntax tree. Third, the XSS exploit code can be partial or mixed up with other irrelevant text. We proceed and present all these issues in detail in the following section. For each problematic case we discuss how xHunter is enhanced to solve each issue.

6.3 Challenges

We now proceed and present the three challenges that xHunter has to deal with, as were mentioned in Section 6.2: (a) web applications quirks, (b) JavaScript relaxed syntax and (c) exploit isolation. We discuss each of these in detail. We present URL examples taken from XSSed.com for each particular case.

6.3.1 Web Application Quirks

Web applications use URLs for communicating with web browsers. There is a well defined specification for URLs [BLMM94]. However, web applications often use their own encoding scheme or custom delimiters for separating the query string and the parameters from the rest of the URL. There are cases where xHunter can try alternative methods for parsing a URL. For example, a significant amount of URLs contained in the XSSed.com repository use “"”, instead of “?” for separating the query string from the URL. In other cases there is no special delimiter used at all, thus xHunter, inevitably, scans the whole URL for JavaScript injections.

There are cases which xHunter cannot handle. These are cases where the URL is encoded using a custom scheme by the web application. In Figure 6.3, we present two example URLs,

```
[http://xssed.com/mirror/55309]

http://www.metacrawler.com/
metacrawler/ws/results/Web/
!3Cscript!3Ealert(!2FXthe _
  miller!2F)!3C!2Fscript!3E/1/41

[http://xssed.com/mirror/64043]

http://www.turktelekom.com.tr/tt/
portal/!ut/p/c0/XYzBCoJAFEX_RQhq
9Z5a0OoEI..RshwIQj/
```

Figure 6.3: Custom encoding used by web applications. In the second example some parts are omitted for better presentation.

which contain exploit code. In the first case a similar, but not identical, to URL encoding is used (the character `!` is used instead of `%`). This case can be handled if the scheme is used frequently (we found just a few cases in the sample collected from XSSed.com). The second case is impossible to handle, since the web application encodes all URLs in a scheme known only by itself. Hopefully, in more than 10,000 of URLs collected from XSSed.com there are only a few cases that follow this paradigm.

6.3.2 JavaScript Relaxed Syntax

JavaScript has a very relaxed syntax. It is quite possible for a text fragment to have a valid JavaScript syntax tree. For example, consider the two expressions in Figure 6.4. They are parts taken from a sample of benign URLs and both of them produce valid syntax trees with high depth. However, these expressions are not JavaScript exploits. xHunter employs two techniques in order to deal with such cases: (a) the reverse code heuristic and (b) weighted parse nodes. We proceed and analyze both techniques.

Reverse Code Heuristic

Expressions like the first one listed in Figure 6.4 have the following properties. First, they can be parsed from left to right and from right to left. Second, they produce a syntax tree with the same depth, no matter the parsing direction. On the other hand, JavaScript exploit code is highly unlikely to produce even a valid syntax tree if parsed from right to left. Thus, xHunter whenever finds an expression that produces a valid syntax tree, attempts to parse the expression *reversed*. The expression is not considered suspicious if the reversed expression produces a syntax tree with the same depth. This heuristic solves cases in the sample of URLs collected from XSSed.com, which otherwise would be considered as false positives.

Weighted Parse Nodes

Parse nodes which refer to tokens such as "`.`" (`DOT`) and "`+`" (`PLUS`), for example, can be repeated several times in an expression and thus result to parse nodes that contribute to the final syntax tree's depth. These tokens occur frequently in URLs, without being part of a JavaScript program. On the other hand, there are tokens that are more likely to be part of valid JavaScript code, such as the `LP` token, which denotes a left parentheses occurrence. These tokens occur less frequently in URLs and much more frequently in JavaScript code.

xHunter assigns a weight to each parse node. The overall depth of the parse tree is a weighted contribution of all parse nodes. There are nodes that have no contribution at all (such as the `DOT` token), nodes that have negative contribution (such as the `NAME` token) and nodes that have high contribution (an `LP` token has double contribution).

6.3.3 Exploit Isolation

It is highly likely that a JavaScript exploit is not isolated in a URL parameter. For example, consider the example in Figure 6.5. The value of the first query string parameter is `";alert(document.cookie)//`. This fragment does not parse as is. However, when the target web page is rendered the fragment is attached to an existing JavaScript expression and the exploit code runs. Multiple parsing attempts must be carried out, removing characters from left and right until a valid JavaScript expression is isolated, in order to detect this code. xHunter starts parsing from left to right. If the result does not produce a valid syntax tree, xHunter reduces the parsing window by removing a character from the left. If all characters are consumed, then xHunter reduces the whole expression by removing a character from the right, the window is set to the initial size (minus the removed character) and the whole

```
foo;1,2,3,4,5

LC:
 SEMI:
  NAME:
 SEMI:
  COMMA:
   NUMBER:
   NUMBER:
   NUMBER:
   NUMBER:
   NUMBER:

id=331653;t=49;l=1

LC:
 SEMI:
  NUMBER:
 SEMI:
  ASSIGN:
   NAME:
   NUMBER:
 SEMI:
  ASSIGN:
   NAME:
   NUMBER:
```

Figure 6.4: JavaScript has a relaxed syntax. Even simple text fragments can produce a syntax tree with high depth.

process restarts. This strategy results in a high computational overhead. However, as we have already stated, xHunter is not meant to be a defense mechanism against XSS attacks that needs to run in real-time. xHunter is designed to process large web traces for exporting statistics related to XSS attacks.

6.4 Case Study

In order to perform a preliminary evaluation of xHunter we test the tool with two samples of URLs. The first one is collected from XSSed.com, the largest XSS repository with public access. It includes about 11,000 URLs which contain XSS attacks and target real web sites.

```
[http://xssed.com/mirror/65494]

http://www.economie.gouv.fr/
recherche/lance_recherche.php?
mot=";alert(document.cookie)//
&search_go=ok
```

Figure 6.5: XSS code can be found partial in URLs. In this example the “"” character (double quotes) must be omitted in order for the rest of the code to produce a valid syntax tree.

The second sample contains 1,000 URLs collected from a monitoring point of an educational organization with about 1,000 users.

6.4.1 XSSed.com

XSSed.com is a public XSS repository. It contains about 11,000 XSS attacks as of March 2010. The repository classifies all attacks in two categories, namely “XSS” and “Redirection” (or “Frame Redirection”). The “Redirection” category involves URLs that, when clicked, perform a redirection from the target web site to a web site of the attacker’s choice. However, we found many cases that were misclassified. Many URLs that perform a redirection are listed under the “XSS” category. xHunter has not been designed to detect redirection or `iframe` injection. Unfortunately, there is no way to filter out all these redirection URLs from the original set.

In Table 6.1 we list properties of the URLs marked as suspicious. Observe, that xHunter succeeds in identifying 8,204 (out of 10,535) URLs, which are known to be XSS exploits. The most popular exploitation technique is by using the `alert()` function. However, as we have already stated, XSSed.com is a repository holding proof-of-concept attacks and not real ones. Nevertheless, there are also other popular JavaScript constructs used, like `document.cookie`, `document.write()` and `String.fromCharCode()`.[3] Finally, xHunter records 274 cases where none of the above constructs is used. For example, there are cases where the exploit code contains: `<img onload=xss()>`.

In Table 6.2 we list properties of the URLs marked as clean. Overall, xHunter marks

[3]These categories overlap. For example it is possible for a URL to contain the code: `alert(document.cookie)` or `document.write(document.cookie)`.

Occurrences	8,204	77.8%
`alert()`	7,895	74.9%
`document.cookie`	1013	9.6%
`String.fromCharCode()`	550	5.2%
`document.write()`	50	0.4%
Other	274	2.6%

Table 6.1: XSSed.com sample. URLs marked as suspicious.

Occurrences	2,331	22.1%
Redirection	611	5.7%
`iframe`	292	2.7%
Redirection and `iframe`	7	0.0%
`<script src="">`	389	3.6%
`<img src="">`	39	0.3%
POST	779	7.3%
Other	268	2.5%

Table 6.2: XSSed.com sample. URLs marked as clean.

2,331 as not suspicious. Since, all URLs are collected from XSSed.com these are false negatives. However, this is not actually the case. The sample contains redirections, `iframes` and `<script>` elements that include JavaScript source code from a third party web site. xHunter has not been designed to deal with these cases. We also include in the trace 779 URLs that use HTTP POST instead of HTTP GET for attacking the web application. XSSed.com provides this information so we can distinguish all URLs that use POST from URLs that use GET. The operation of xHunter does not change in the case that has to process a POST request. The task of the tool is easier since the parameters (included in the POST part) are easily isolated from the rest of the URL. However, we leave on purpose all these URLs inside the trace to verify that none of them will be marked as suspicious. Indeed, xHunter successfully marks all 779 POST URLs as clean.

By subtracting all the above cases, 268 XSS exploits remain marked as clean. With manual examination we find out that these exploits are mixed. Some of them are redirection attacks (but classified as “XSS” and not “Redirection”), some other take advantage of web application quirks we discussed in Section 6.3. Thus, xHunter has less than 3.2% false negatives.

6.4.2 Benign URLs

We run xHunter with a second trace, which contains 1,000 URLs collected from a monitoring point in an educational organization with about 1,000 users. All URLs are considered benign. xHunter marks 20 URLs as suspicious. That is, xHunter has about 2% of false positives. We manually examine the 20 false positives. All 20 cases are URLs that, instead of `‘‘&’’`, use the `‘‘;’’` character as a delimiter for query parameters. The `‘‘;’’` character is significant in JavaScript. xHunter can be easily enhanced to treat both `‘‘;’’` and `‘‘&’’` characters as delimiters that separate parameters in a query string. We modify the tool and we rerun all experiments. The 20 URLs in the benign trace are not marked as suspicious. The results for the trace collected from XSSed.com are not altered by this modification.

Chapter 7

XSS Study

7.1 Introduction

Despite the significant effort by the academic community towards a solution that aims at protecting web applications and end-users from web attacks [JSH07, NSS09, GC09, Sek09, TLV09, APK$^+$10] and the various available browser-based solutions [Rei08, Mao06], the landscape of web exploitation imposes a real threat. Consider, that according to a Symantec report published in 2008 [Sym], the second half of 2007 was dominated by incidents related to Cross-Site Scripting (XSS) exploitation. Moreover, according to a September-2009 report published by the SANS Institute [SAN] attacks against web applications constitute more than 60% of the total attack attempts observed on the Internet and web application vulnerabilities account for more than 80% of the overall vulnerabilities being discovered. To make things worse, there is evidence that modern web attacks are targeting the high ranked web sites, which accumulate millions of Internet visitors, such as Twitter [XSSc] or web applications operated by the top leading industry vendors [mca].

There is an ongoing arms race between technologies that aim at providing countermeasures against web exploitation and attackers inventing more elaborate exploitation techniques. In order for the security researchers to evolve the state-of-the-art of the tools and technologies for defending against web attacks, we need to understand the already published incidents associated with web exploitation. For the first time, we present an extensive analysis of a large repository documenting about 12,000 samples of real-world web attacks.

Methodology. We crawl the most well known repository hosting web attacks, XSSed.com [FP], and we collect about 12,000 unique exploits. All these exploits are related to real incidents of web exploitation over the last three years. Each exploit is characterized by a set of metadata properties, generated by the repository (such as the *Target* web site, the *Submission* and *Publication* date, etc.). We further process each exploit with a custom URL processor [AKM10] and derive even more details about the exploitation techniques used (if it is a `<script>` or `<iframe>` injection, if the exploit is transmitted over an encrypted communication channel, etc.). We finally present a series of statistics and highlight various sets of properties that form a better picture of the web attack landscape.

Contribution. The contribution of this chapter is the following.

1. For the first time, we present an analysis of a large set of about 12,000 real-world web exploits collected by a well known repository, XSSed.com [FP]. Our analysis has some very interesting results. A few representative of them are the following. About 8% of the web exploits are targeting web sites that communicate over HTTPS, giving the illusion of a secure communication to the end-user. About 8% of the targets are highly ranked web sites, according to Alexa.com (Top-1000). It takes a few to hundreds of days, even for the Top-100 web sites, to patch the published vulnerabilities.

2. We identify the dominant patterns of web exploitation. Attackers first attempt to exploit a vulnerable web site by trying *common attack recipes*. These attack recipes serve as a proof-of-concept (PoC) exploit. It is highly likely that such a PoC exploit is published by XSSed.com. By studying over 12,000 web exploits we can derive patterns associated with the most common attack recipes. These patterns can be useful to the IDS community [R$^+$98, Pax99].

3. We process all web exploits with a custom URL processor [AKM10] and show that there are exploits that cannot be captured by static filters [BBJ10a]. We further analyze the JavaScript syntax trees of the web exploits, in the cases that it is applicable, and show how a sophisticated detector can capture an even broader family of exploits. We derive

a set of 18 signatures based on JavaScript syntax trees, which can successfully detect 77% of all exploits that cannot be detected using static filters.

Organization. The remaining chapter is organized as follows. In Section 7.2 we present in details the methodology and the tools used for the data collection. In Section 7.3 we present the results of our study and in Section 7.4 we expand our thesis towards static filters by arguing that detectors can be enhanced with JavaScript syntax-tree signatures for capturing a broader family of web exploits.

7.2 Methodology

In this section we present the tools and techniques we employ to acquire the data. We argue about the reason we select the particular repository used in this study. We further outline the process of exporting additional metadata from the data collected with the assistance of a custom URL processor [AKM10]. We finally present a short introduction over the inherent properties and additional metadata of the data sample.

7.2.1 Overview

We collect all data from the public repository XSSed.com [FP] using a custom Python crawler. This particular repository is well known in the web security community. In fact, it has been used in the past for evaluating a number of proposed anti-XSS frameworks such as DSI [NSS09], xJS [APK$^+$10] and XSSAuditor [BBJ10a]. There are a few other attack sources, such as Bugzilla [SC05], the Web Hacking Incident Database (WHID) [whi] and Zone-H [zon]. Bugzilla is probably the largest open source repository hosting application vulnerabilities. Bugzilla is not suitable for our study, since it does not specialize in web exploitation. Among the thousand vulnerabilities of desktop applications, it lists only exploits associated with the top web applications, such as Twitter or Facebook, or exploits for generic web software that is employed by many web sites. Our study aims at covering *every-day* incidents associated with web exploitation. WHID specializes in web exploitation, but covers only incidents that have massive media exposure. For example, the latest report of WHID covers only 150 incidents

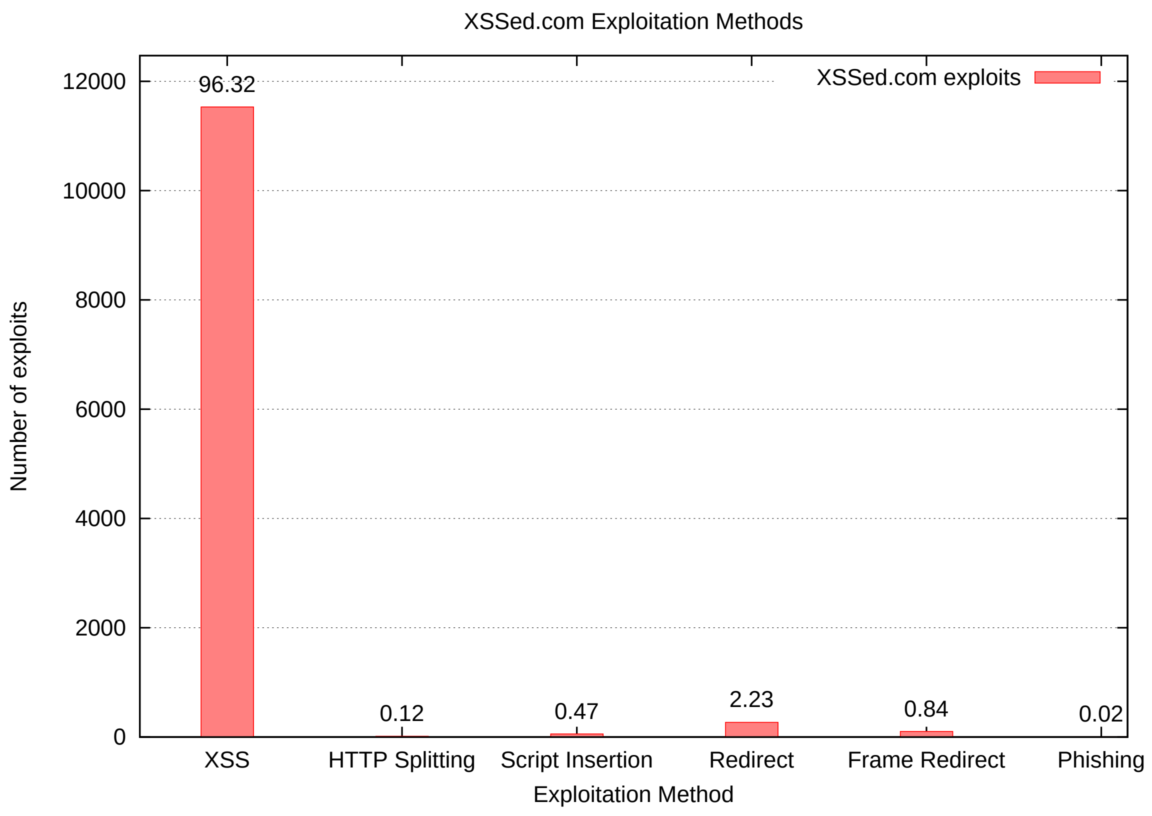


Figure 7.1: The six different categories used for exploit classification by XSSed.com.

for the entire 2010. Finally, Zone-H provides information about *web site defacement*. This is a particular method of showing and proving that a web site is vulnerable, by changing the web site's initial web page. The defacement itself can be carried out with many different techniques: from compromising completely the web server to exploiting a directory traversal vulnerability of the web server. Thus, recorded defacements provide little information about the exploitation techniques used by the attackers. In Section 7.3, we attempt to identify if there is strong correlation between the incidents recorded by XSSed.com and the defacements recorded by Zone-H.

XSSed.com hosts over 12,000 of web injections classified in six different attack flavors. In Figure 7.1 we present a histogram with all exploits categorized according to XSSed.com. The majority of the exploits are XSS injections. However there are some fractions characterized as *HTTP Parameter Splitting*, *Script Insertion*, *Redirection*, *Frame Redirection* and *Phishing*. In Section 7.3, we provide a more descriptive classification of all attacks in the collected sample.

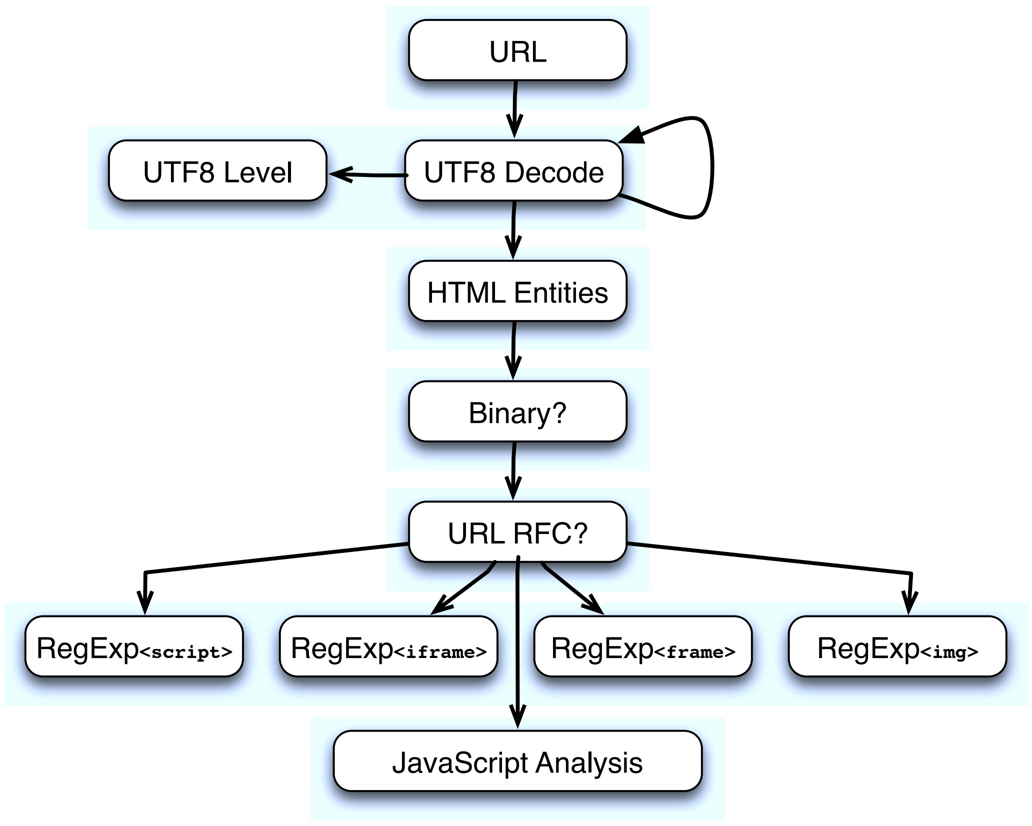


Figure 7.2: Schematic diagram of all steps followed by xHunter for extracting additional information for each web exploit.

In Table 7.1 we list all properties provided by XSSed.com for each published incident. We further process each case with a custom URL processor, xHunter [AKM10]. xHunter takes as input a URL and tries to locate fragments that can produce a valid JavaScript syntax tree. In parallel with this process, xHunter exports additional details, which further provide a more descriptive collection of information for each one of the exploits. We list all properties exported by xHunter in Table 7.2.

7.2.2 URL Processing

We now present in more details all steps followed by xHunter for exporting all additional information listed in Table 7.2.

xHunter initially considers that the input is encoded in UTF8. There are many different representations for a given character in UTF8 and this is one of the dominant reasons that

Data	Description
ID	The ID of the web exploit.
URL	The complete URL needed for a PoC exploitation.
Category	The category as classified by XSSed.com (see Figure 7.1).
Date Submitted	The date the web exploit submitted to XSSed.com.
Date Published	The date XSSed.com manually verified the exploit and published it.
Date Fixed	The date the vulnerable web site was patched.
Status	FIXED if the vulnerability has been fixed. UNFIXED otherwise.
Author	The author of the web exploit.
Domain	The target domain.
Page rank	The page rank of the target domain according to Alexa.com.
POST data	The POST data required if the attack exploits a POST form.

Table 7.1: All information provided by XSSed.com for each web exploit.

the community considers hard to develop accurate filters against XSS. A rich collection of these representations can be found in the *XSS Cheat Sheet* [Han]. xHunter takes into account *all* possible variations. Furthermore, xHunter applies the UTF8 decoding process until the URL remains intact in decoding. Note, that there are cases in our sample, where a URL has been partially encoded in UTF8 twice or more times. This is obviously another obfuscation trick employed by attackers to evade detection. We use the term *UTF8 Level* to describe this behavior (see Table 7.2).

Furthermore, the tool compiles all HTML entities, such as `&amp;`, `&gt;` and `&lt;`, to their textual representation. It also checks if the URL contains characters outside the ASCII printable range and if it follows the URL RFC [BLMM94]. URLs following the RFC have the form of:

Property	Description
HTTPS	True if exploit is transmitted over HTTPS.
Binary	True if exploit contains binary characters.
Malformed	False if URL follows strictly the RFC [BLMM94].
Has Scripts	True if exploit contains `<script>` elements.
Has iFrames	True if exploit contains `<frame>` elements.
Has Frames	True if exploit contains `<iframe>` elements.
Has Images	True if exploit contains `<img>` elements.
HTML entities	True if exploit contains HTML entities likes `&amp;`.
UTF8	True if exploit contains any of the possible UTF8 encodings [Han].
UTF8 Level	The number of levels of UTF8 encoding.

Table 7.2: Additional information provided by xHunter for each web exploit included in the repository of XSSed.com.

```
protocol://domain/query?p1=v1&p2=v2...&pn=vn
```

Modern web applications frequently use their own URL schemes, for reasons which are not publicly documented, making the task of analyzing a URL quite hard. All web exploits that do not strictly follow the URL specification are marked as *Malformed*. For all URLs, malformed or not, xHunter applies a series of regular expressions, in order to locate possible `<script>`, `<iframe>`, `<frame>` and `<img>` elements. These are the most common HTML elements used for injecting foreign code in a web application. We also locate any markup elements except the aforementioned ones. Regular expressions are not considered safe for capturing web attacks [BBJ10a]. However, there are plenty of widely used products [Mao06, Rei08], which base a significant part of their functionality in a set of carefully selected regular expressions. xHunter relies on regular expressions in part, since its core operation is given a text fragment

UTF8 Level	Exploits (%)
No UTF8	3461 (28%)
One Level	8509 (71%)
Two Levels	121 (0.01%)
Three Levels	8 (0.0006%)

Table 7.3: Exploits that use UTF8 encoding, sometimes multiple times, in order to evade detection. Levels indicate how many times a fraction of a URL must be UTF8 decoded in order to have ASCII text.

to export possible JavaScript syntax trees. We rely on regular expressions just for producing statistics and not for preventing or detecting exploits.

Finally, xHunter tries to locate fragments in the URL or in the data of the POST part of a web exploit, that can produce a valid JavaScript syntax tree. We depict schematically the complete operation carried out by xHunter in Figure 7.2.

7.3 Experimental Results

In this section we present the results of an extensive analysis of about 12,000 web incidents recorded by XSSed.com. For better presentation, we divide the section in three families of results. First, we present an analysis about generic properties of web exploits. Second, we present various characteristics about the web sites that are the targets of the attacks. Finally, we give a short analysis regarding the quality of the repository.

7.3.1 Exploitation Method

One of the fundamental characteristics of a web exploit is the protocol and HTTP method used to carry out the attack. In Figure 7.3 we depict the protocol, HTTP or HTTPS, and the HTTP method, GET or POST, used by every web exploit in the data set. As expected, the majority of the exploits are transmitted over HTTP and use the GET method to perform the injection. However, it is interesting to note the following. First, there are exploits which are communicated to the vulnerable web server over HTTPS. Although, the fraction is low, less than one in ten incidents, it is still worth noticing. HTTPS is considered by many users as *secure*, in terms that a third party cannot eavesdrop or steal the user's data. This belief is

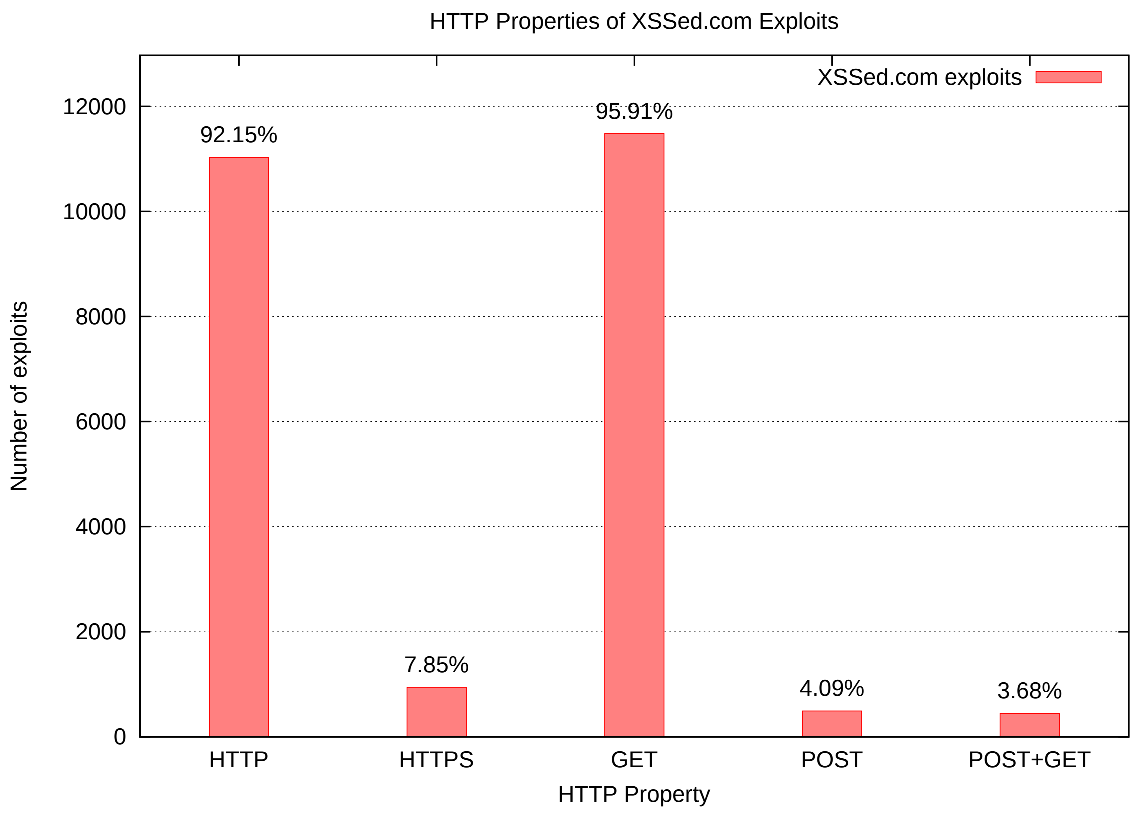


Figure 7.3: The HTTP property (protocol and method) used by each web exploit.

heavily driven by many browser indicators, such as padlocks or officially signed certificates, carefully designed to draw the attention on the user that the web site is trusted [SEA+09]. However, a vulnerable in code-injection attack web site is still exploitable no matter the encryption in the communication channel. Nevertheless, we expect that web sites served in HTTPS invest, in general, more in security, which requires significant financial resources and thus the percentage of incidents is low.

As far as the HTTP method is concerned, as expected most of the web sites have vulnerable HTML forms, which are submitted using the GET method. POST method is used more often for submitting information for further processing and apparently web developers treat more carefully the respective code. It is still worth noticing that almost all incidents exploiting a POST form are based on URLs that include GET parameters. The HTTP specification does not prohibit this behavior, although the intuition behind this programming habit remains undocumented.

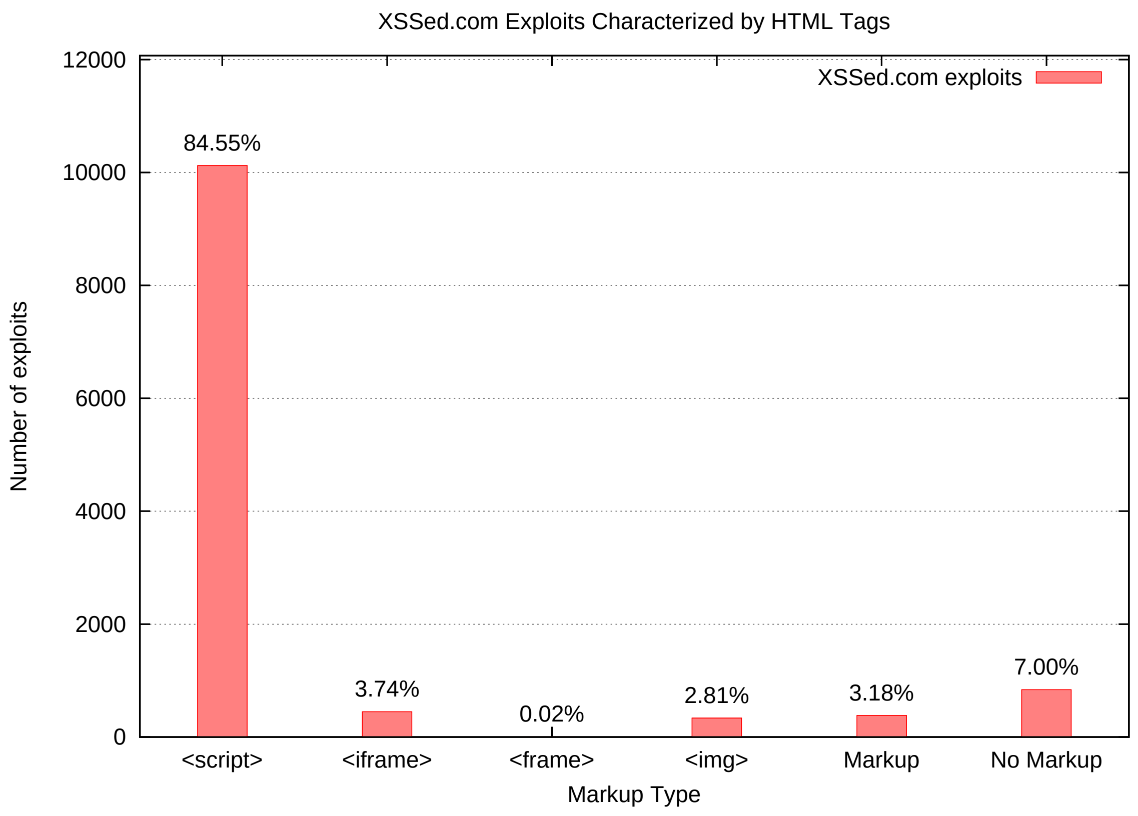


Figure 7.4: HTML markup used in web exploits.

Many web exploits inject HTML tags, such as `<script>` or `<iframe>`, in vulnerable web applications. This is evident in Figure 7.4 where we present all exploits according to HTML tags they use (in some cases multiple markup elements are used). As expected, the majority of all exploits, about 85%, is based on the injection of a `<script>` tag. The percentages of exploits injecting `<frame>`, `<iframe>`, `<img>` or any other markup element [1] is quite lower. However, the interesting part is that nearly one in ten exploits does not use any markup at all. This is very crucial for anomaly detectors that are based purely on static signatures and regular expression [R$^+$98, Rei08, Mao06]. We proceed and collect all exploits that do not use any markup and perform a JavaScript analysis using xHunter. We present our results in Section 7.4.

Finally, one interesting finding is the use of multiple layers of UTF8 encoding. Encoding URLs in UTF8 is typical. However, web exploits can take advantage of the encoding in

[1] It is worth pointing out that there are plenty of exploits that inject generic HTML markup elements. These elements can be the very common, like `<strong>`, or even deprecated ones, like `<marquee>`.

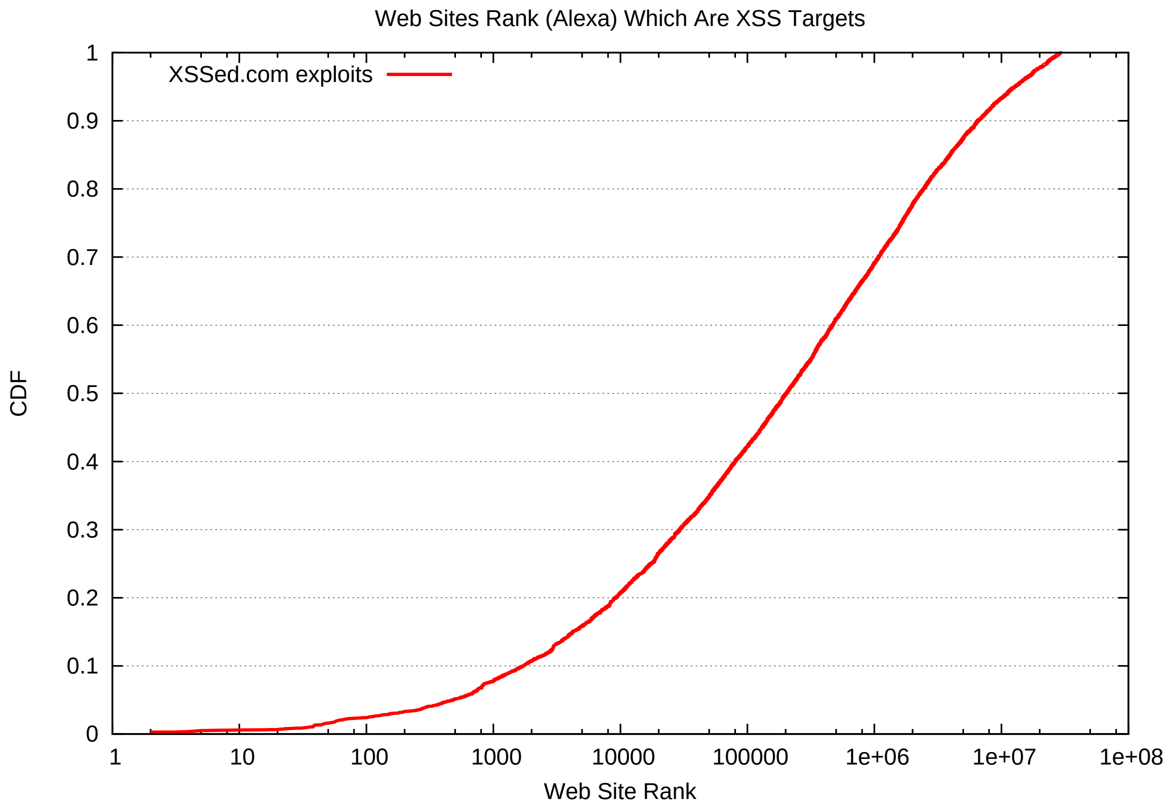


Figure 7.5: The rank, as provided by Alexa.com, of web sites that are targets of XSS attacks.

order to evade detection from a passive monitor, an IDS or a filter implemented in the web application. In Table 7.3 we list the percentage of exploits that use UTF8 encoding. The majority of all exploits use UTF8 encoding, about 71%, but there are also just a few that use multiple layers of UTF8 encoding. A UTF8 Level three indicates that the URL has to be UTF8 decoded three times in order to become intact in further decoding.

7.3.2 Target Characterization

We now investigate the characteristics of the targets which experience web attacks. In Figure 7.5 we plot the CDF of all web sites' rank, as provided by Alexa.com. Observe that attackers do not seem to be very selective in choosing their victims. About 3% of all victims can be considered very popular, since they are listed in the Top-100 of Alexa's ranking. About 8% can be considered to have average popularity, since they are listed in the first one thousand of Alexa's ranking. The remaining is composed by a variety of different web sites'

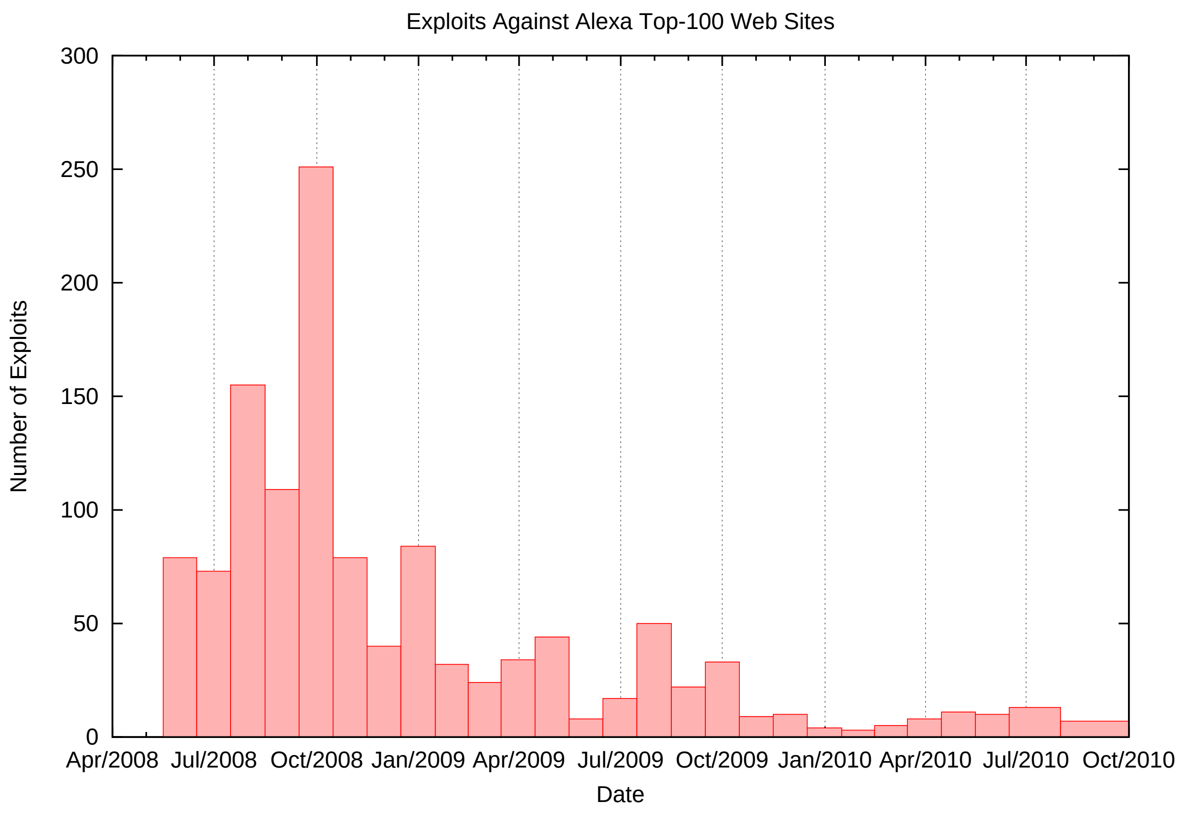


Figure 7.6: XSS exploits during the last couple of years against all Top-100, based on Alexa.com, web sites. Notice the increase in 2008, which is also reported by Symantec [Sym].

popularity classes. There are web sites that are listed in the first few thousands of Alexa's entries and web sites that can be hardly classified as known, since their ranking is over a few millions. The intuition is that a broad class of different web attackers exists. All attackers do not share the same incentives. Some of them target the very prestigious web sites and some others *all* vulnerable ones, no matter if they serve a very limited user base.

In Figure 7.6, we plot all victim web sites from the Alexa's Top-100 for the last couple of years. For each exploit we record the date the exploit was submitted to the repository. Notice the increase of reported exploits in 2008. This incident has been also reported by Symantec [Sym]. In recent days, the rate of exploit submission has been decreased. However, this rate refers to the Top-100 web sites, which usually have significant financial resources, in order to make use of the latest security technologies, hire penetration testers, and, in general, invest on security features. It is worth noticing that there are still a few reported vulnerability

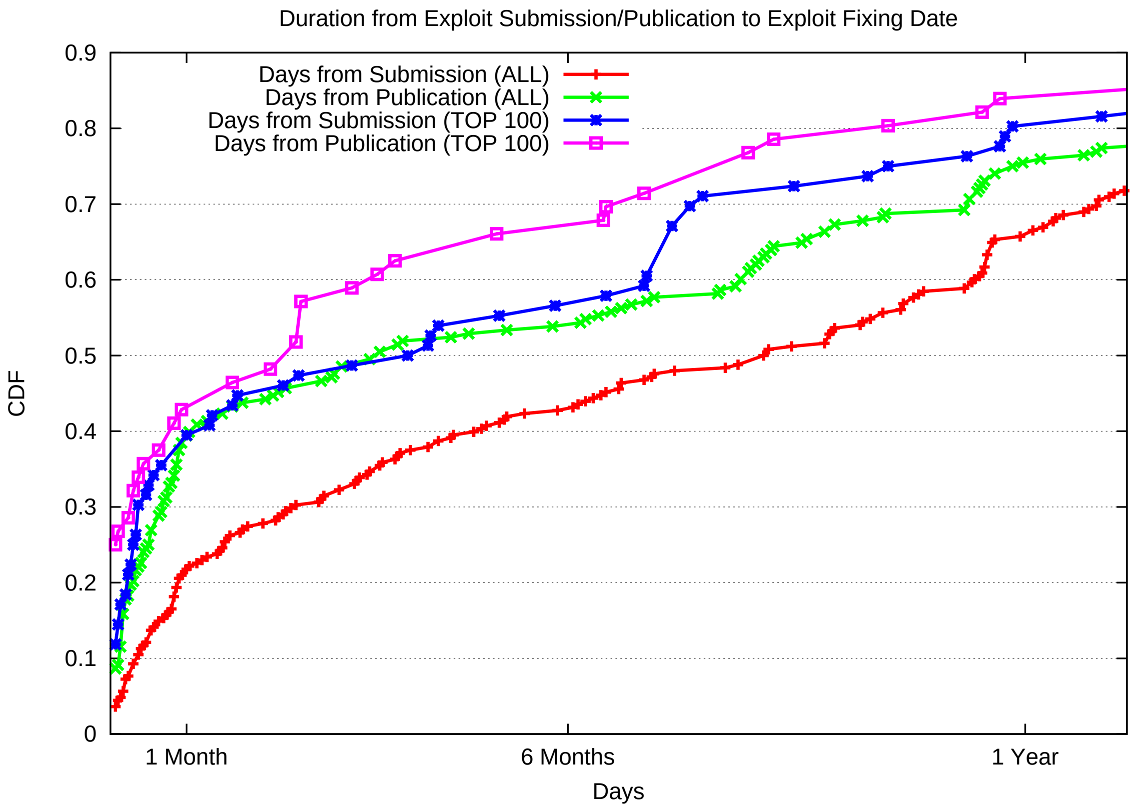


Figure 7.7: Days required by a web site to repair the vulnerability after the submission or publication date of an exploit. We depict two sets of web sites, the ones that are ranked in Top-100 by Alexa.com and all the others. Notice, that, in general, the publication date is closer to the date of patching than the submission one. Also, Top-100 web sites seem to be faster in patching.

per month for this class of web sites.

Another interesting aspect we investigate is the period it takes for a web site to repair the vulnerability. In our sample, for each incident we have recorded three dates: (a) the submission date, which specifies the date the exploit was submitted to the repository (b) the publication date, which specifies the date the exploit was published in the repository and (c) the date the repository was notified that the vulnerability has been repaired. In Figure 7.7, we plot the days required by a web site to repair the vulnerability. We divide all exploits in two classes: (i) exploits associated with Top-100 web sites and (ii) all others. We also plot the days required for repairing the vulnerability as counted from the submission and publication date, respectively. Notice, that the the publication date is closer to the date of patching than the submission one. Also, Top-100 web sites seem to be faster in patching. However, even

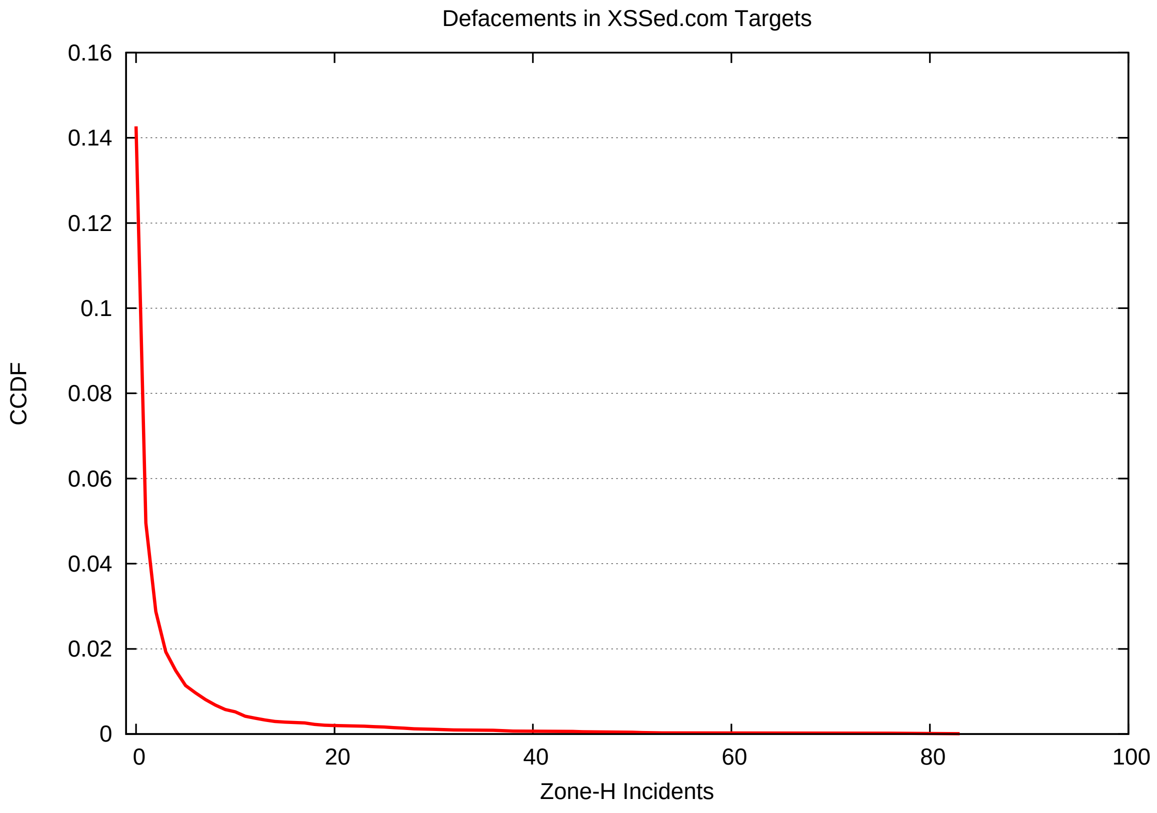


Figure 7.8: Web sites listed in XSSed.com as attack targets and have been also experienced defacement incidents as reported by Zone-h.org. At least one defacement incident has been experienced by 14% of the XSSed.com sample.

the Top-100 web sites require from a few days to a few months to repair a problem. This can be attributed to the following reasons. First, some vulnerabilities are critical and they are repaired in the same day, or at least in a few days. Second, some vulnerabilities may not be that critical and they take a longer period of time to get fixed. Third, some web sites may repair the vulnerability without communicating it to the repository.

Moreover, we explore whether the victims of a web attack are also the victims of *defacements*. A defacement is an action where an attacker changes the content of a web page, with a message of her own. Usually, this message is connected with a protest, or it is simply used as a proof of the technical skills of the attacker. However, the conditions required to achieve a defacement are not clear. There are cases of defacements that require very limited technical knowledge. Since a defacement is not directly connected with earning of money, it is more likely to be carried out easier than an XSS or CSRF attack. Zone-h [zon] is a public

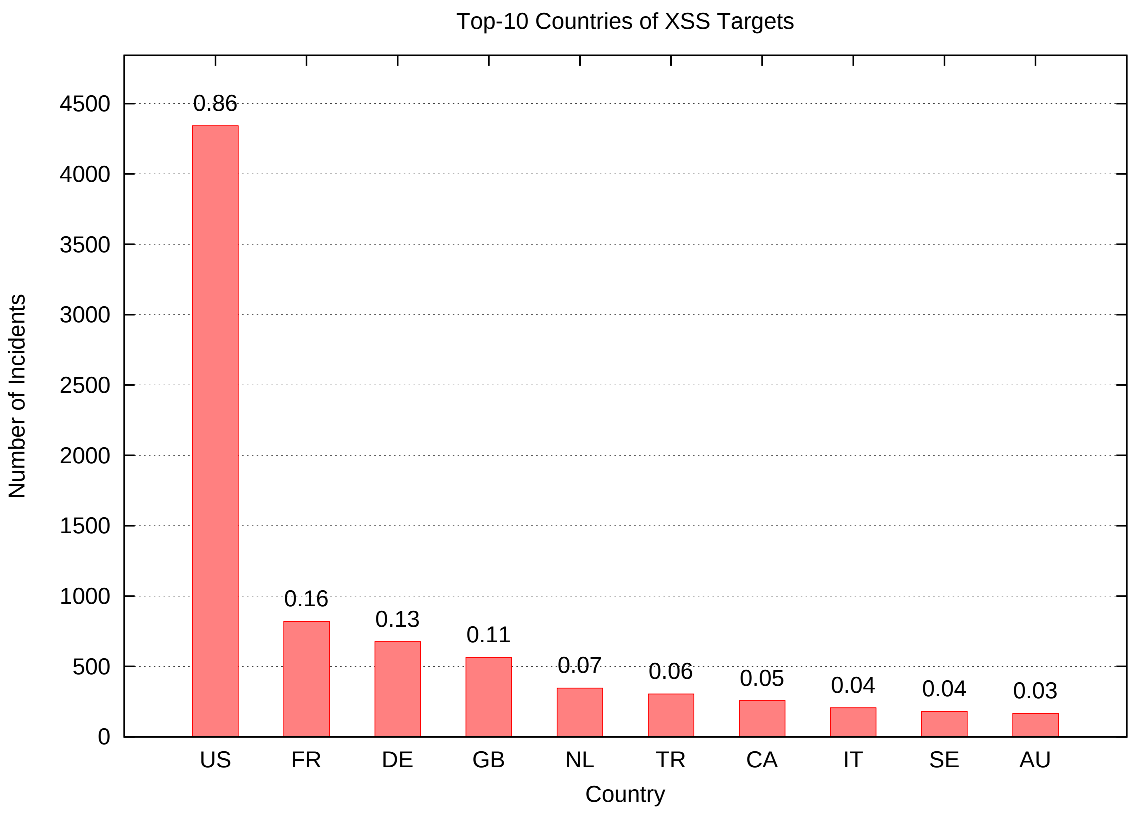


Figure 7.9: The Top-10 countries that host the most frequently attacked web sites.

repository that hosts more than 75,000 defacement incidents. Notice, that at the same time this repository hosts seven times more incidents than XSSed.com. For each web attack in our sample taken from XSSed.com, we query Zone-h to see if the same domain has experienced a defacement. Nearly 15% of domains that have experienced a web attack, have been also experienced *at least one* defacement. This leads us to conclude that defacements and web attacks, like XSS, are not highly correlated. In Figure 7.8 we plot the Complementary CDF (CCDF) of defacement incidents that have been experienced by web sites reported in XSSed.com. Most web sites have experienced one defacement incident and a few of them have experienced a few tens of defacement incidents.

Finally, in Figure 7.9 we list the Top-10 of the most attacked countries, obtained using `geoiplookup` [Max06]. Observe that the majority of domains that are listed in XSSed.com is hosted in the United States and even the rest of the Top-10 is filled up with European countries. This may be due to several reasons. First, the repository may be not well-known

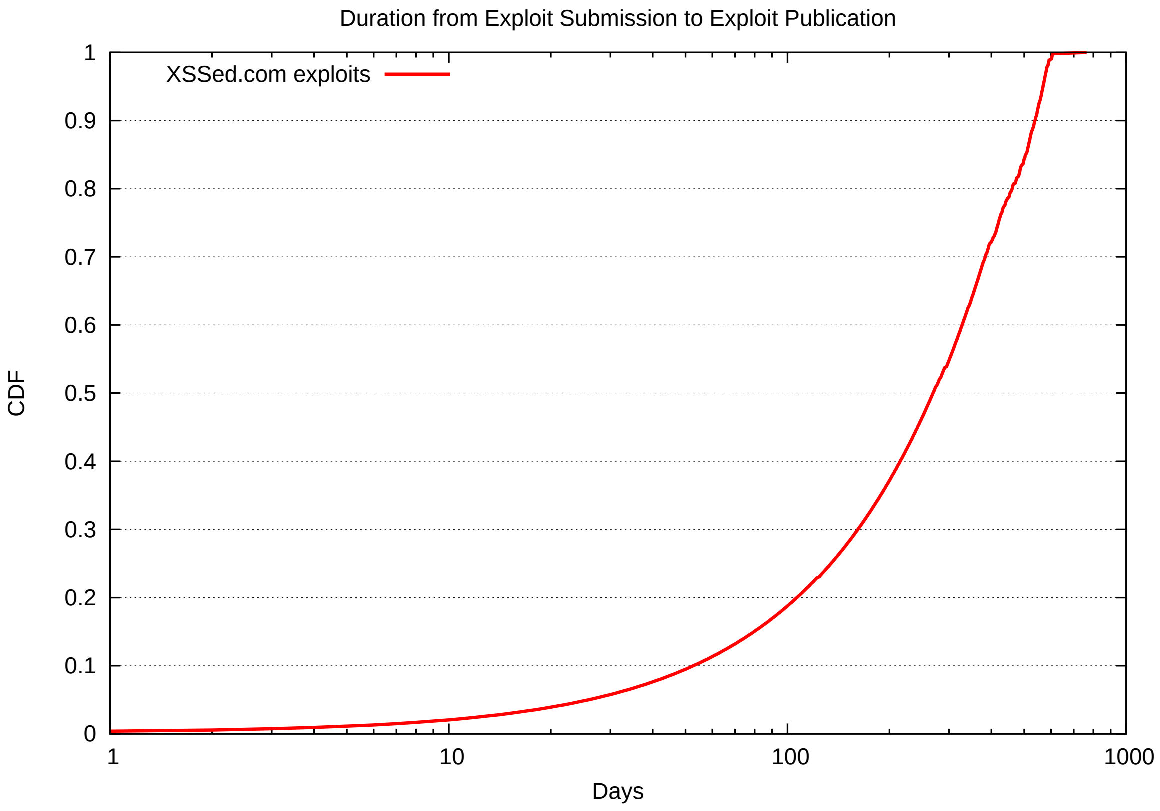


Figure 7.10: Days required for XSSed.com to verify and publish a submitted web exploit.

outside of the United States and Europe. Notice, that according to `geoiplookup` XSSed.com is located in France. Second, this might be evident that web exploitation is most frequently occurring in the west world. Consider, that there are attack flavors, which are launched using the web, that have arise in Asian countries but never resulted in any reported incident in the United States/Europe [CYK10].

7.3.3 Repository Characterization

Finally, in this section, we investigate how fast the repository acts upon new attack events. In Figure 7.10 we present the days required for XSSed.com to verify and officially publish a reported incident. Observe that half of the reported incidents have been published from a few days to a few weeks upon reporting and half of them have been published months after reporting. Moreover, in Figure 7.11, we plot the amount of published exploits that have been submitted each date. Observe, the bursty behavior, which suggests that submitted

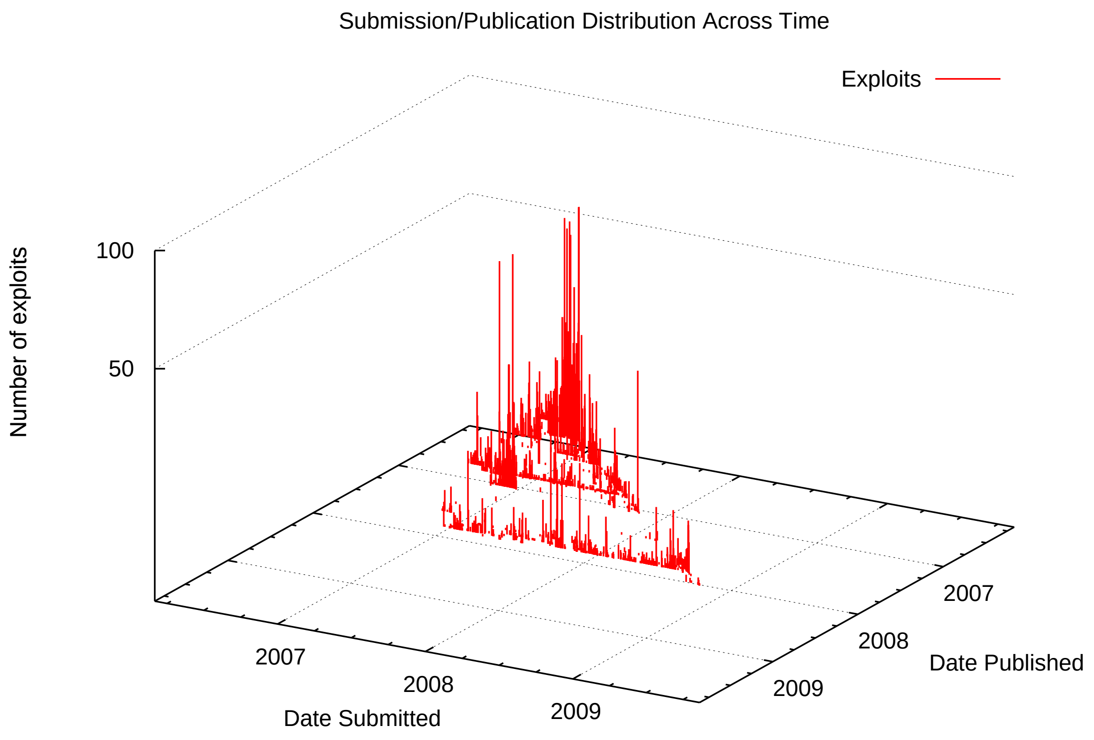


Figure 7.11: Number of exploits that have been submitted and published each date.

exploits are processed in a batch fashion. This is evident, since in order for an exploit to be published, manually inspection and testing is required. Also, notice that more recent exploits are processed in smaller batches. Apparently, in recent years the repository experiences better maintenance, most probably because of the publicity it started to receive. However, we can only speculate for this, since we have not come in contact with the repository's maintainers, in order to eliminate any bias produced by any communication that can affect the results of this work.

7.4 JavaScript Analysis

In Section 7.3 we conclude that about 7% of all web attacks reported in XSSed.com contain no markup, such as `<script>`, `<iframe>`, `<frame>`, `<img>` or any other HTML element (see Figure 7.4). Exploits that do not use markup are hard to be captured by tools that are based

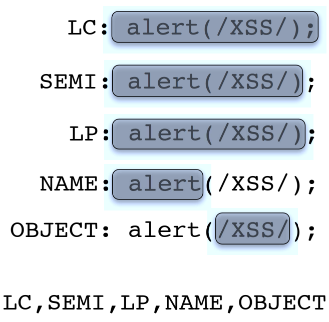


Figure 7.12: An example of how a JavaScript syntax-tree is extracted from an expression.

on static signatures and regular expressions [Mao06, Rei08, R+98]. This is because it is hard to express all JavaScript grammar using regular expressions. In this section we take the 7% of URLs that contain no markup and we try to identify if they embed parts that can produce a valid JavaScript syntax tree.

7.4.1 Results

A JavaScript syntax tree is composed by a series of tokens that have special semantics in JavaScript. For example, consider the JavaScript syntax tree, which is depicted in Figure 7.12. The tree is composed by the tokens `LC`, `SEMI`, `LP`, `NAME` and `OBJECT`. This series of tokens identifies the JavaScript program `alert(/XSS/);`. The fragment of the program that maps to each token is also highlighted in Figure 7.12. Notice, that each token does not map directly to a JavaScript entity. For example, tokens `SEMI`, which is a shorthand for *semicolon*, and `LP`, which is a shorthand for *left parenthesis*, map to the same portion of text. Thus, it is better to think tokens as different *states* of the JavaScript parser.

The fragment of 7% of all exploits that contain no markup is translated to 838 web attack incidents. From these 838 incidents we exclude all of the reports that are not categorized as XSS (see Figure 7.1). We do this in order to filter out incidents that are not directly related with JavaScript injections (redirections, phishing reports, etc.). Thus, we remain with 405 exploits, which (a) do not use any markup, and (b) perform an XSS attack. We perform a JavaScript analysis in all these 405 exploits. We manage to identify 321 cases of URLs, which contain a fragment that can produce a valid JavaScript syntax tree, like the one depicted in Figure 7.12. The remaining cases are consisted by URLs that use special encoding

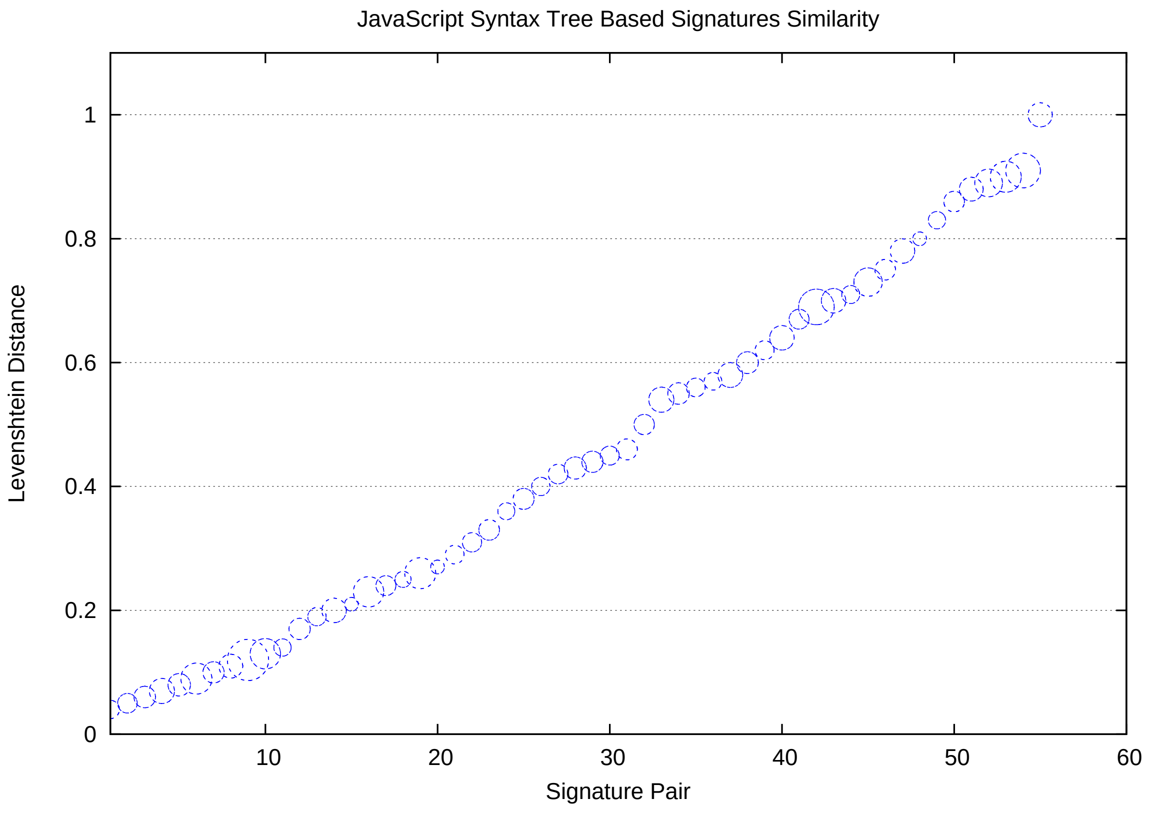


Figure 7.13: Similarity of all JavaScript syntax tree based signatures. The size of the radius represents the Least Common Subsequence (LCS) of a pair of signatures.

schemes [XSSd]. In all 321 identified cases, there are 8 false positives. Some text fragments of a URL may contain tokens, such as `+`, `-` and `/`, which can force the JavaScript parser to produce the syntax tree of a valid JavaScript arithmetic expression. The 313 identified exploits correspond to 77% of all examined cases. Thus, using JavaScript analysis we are able to identify 77% of 405 incidents that cannot be captured by a signature-based tool.

7.4.2 Syntax-Tree Based Signatures

The most interesting aspect of the JavaScript analysis is that we can conclude that a JavaScript syntax tree is a convenient form for producing signatures that can describe families of web exploits. Consider that in all 313 cases there are only 60 unique JavaScript syntax trees. From the perspective of JavaScript syntax trees, there is a major overlap in the ways attackers are trying to exploit web applications. With 60 syntax trees we are able to identify

313 web attacks. On the other hand, by investigating the actual injections, we conclude that all 313 incidents correspond to 140 unique JavaScript programs. For example, the program `alert('XSS')` and the program `open('www.attacker.com')` are different, but their syntax trees are exactly the same.

Moreover, these 60 unique JavaScript syntax trees share common tokens in great extent. To show this, we map each token to a unique letter of the alphabet and we construct strings that represent each syntax tree. In Figure 7.13 we present the Levenshtein distance [Dam64] of all possible pairs of JavaScript syntax trees, expressed as strings. Notice, that half of the comparisons result in a high similarity value. Moreover, for each pair the radius expresses the Least Common Subsequence (LCS). Even the pairs of syntax trees that have low similarity have a significant LCS. These facts drive us to create *families* of signatures based on JavaScript syntax trees. We manually group all 60 syntax trees in 18 signatures. We list all signatures in Table 7.4. For each one we present a possible exploit that the signature matches, as well as the number of overall attacks that were matched by the signature. We use the following notation to express each signature: (i) brackets `()` indicate that the token inside is required, (ii) square brackets indicate that the token inside is optional, (iii) vertical bar `|` indicates logical `OR` and (iv) star `*` indicates that the token can be repeated arbitrary times.

7.4.3 Discussion

The JavaScript analysis we present in this section has two main contributions. First, we list a collection of real web exploits, which are based purely in JavaScript and they do not depend on using HTML elements. It is hard for any tool that is based on static signatures and regular expressions to detect such exploits, because there is no finite set of regular expressions that can express all the possible JavaScript programs. The majority of the examples listed in Table 7.4 is based on the usage of `alert()`, which is a function that displays a dialog box with a message inside the web browser's environment. One can argue that by simply searching for an `alert()` expression can sufficiently enhance the range of exploits that can be captured using static signatures. Indeed, many tools [Rei08, Mao06] use regular expressions for capturing URLs that embed `alert()` calls. However, an attacker can use an infinite set of functions to attack a web site. In fact, an attacker can take advantage of all of the existing JavaScript code which is already loaded by a web application [APM09]. Thus, by simply searching for the `alert()` call is not sufficient. On the other hand, the programs we list in Table 7.4 can be valuable for the IDS community, since they can enhance the range of *attack attempts* that an IDS can detect. Second, we create a new set of rules which is not based on regular expressions, but on JavaScript syntax trees. This set of rules can be beneficial for building more sophisticated IDSs, which can be optimized for web attack detection.

Signature	Exploit Example	#
LC SEMI LP NAME (NAME|NUMBER|OBJECT|STRING)	alert(’XSS’);	60
LC COLON SEMI LP NAME (NUMBER|OBJECT|STRING)	javascript: alert(/xss/);	48
LC SEMI ASSIGN NAME LP NAME (NUMBER|OBJECT|STRING [SEMI NAME])	onmouseover = alert(/XSSbySh4v/)	46
LC SEMI ASSIGN NAME LP NAME LP DOT NAME NUMBER*	onerror = alert(String.fromCharCode(120, 115, 115))	41
LC SEMI SEMI LP NAME STRING [SEMI DOT NAME|SEMI NAME|VAR NAME]	;alert(’xss’);//	25
LC SEMI SEMI LP NAME NUMBER [SEMI NAME|VAR NAME]	;alert(1337);	22
LC COLON SEMI LP NAME DOT (NAME|OBJECT)	javascript: alert(/xss/.cookie)	10
LC SEMI LP NAME [LP] DOT NAME (SEMI STRING|NUMBER*)	alert(String.fromCharCode(88, 83, 83))	8
LC COLON SEMI LP DOT NAME STRING*	javascript: window.open(’...’);	6
LC SEMI SEMI LP NAME (OBJECT|SEMI NAME)	;alert(/Jurpie/);//	3
LC SEMI ASSIGN NAME RELOP STRING (DIVOP NAME*|NAME)	onmouseover="alert(123)">HOVER	3
LC COLON SEMI LP NAME LP (DOT NAME STRING|NAME OBJECT)	image: expression(document.write(’XSS’))	2
LC COLON SEMI DOT LP NAME DOT OBJECT	javascript: alert(/XSS/.source).html	2
LC COLON SEMI LP NAME STRING SEMI LP DOT NAME STRING	javascript: alert(’swfXSSed! :P’); window.open(’...’)	2
LC SEMI COMMA (LP NAME NUMBER NAME|LP NAME)	alert(1337),f	2
LC SEMI DIVOP NAME* LP NAME (NUMBER*|STRING)	{...}alert(40017,0423769097);	2
LC SEMI LP DOT NAME STRING	document.write(’xssed by babaconda’)	1
LC SEMI DEC LP NAME*	-xlt_script_xgt_alert(_xps_Lapanzi_xps_)	1

Table 7.4: JavaScript syntax-tree based signatures. For each one we present a possible exploit that the signature matches, as well as the number of overall attacks that were matched by the signature. We use the following notation to express each signature: (i) brackets `()` indicate that the token inside is required, (ii) square brackets indicate that the token inside is optional, (iii) vertical bar `|` indicates logical `OR` and (iv) star `*` indicates that the token can be repeated arbitrary times.

Chapter 8

Network Flow Contracts

8.1 Overview

Over the past few years an increasing number of organizations has started collecting data about how and when people use the Internet. Web servers, for example, customarily keep the IP addresses of all devices which have accessed any of their content. Similarly, VoIP providers keep logs of all calls each of their customers has made, mostly for accounting, performance, and advertisement reasons. In some countries, it has even become mandatory for some organizations, such as ISPs, to keep a copy of all accesses their customers have done over the Internet for a predefined period of time [dat].

Although such Internet data were originally collected for accounting and performance reasons, they have been increasingly requested and used by Law Enforcement Agencies (LEAs) to help the investigation of on-line-related crimes. However, since these data have not been verified by the end-users involved, they may be inaccurate, fabricated, or even complete bogus. Based on such fabricated information an innocent user may be falsely accused (or suspected) of committing an on-line related crime such as accessing a web site which serves illegal content. To make matters worse, a malicious attacker may even deliberately plant bogus evidence in order to falsely accuse an innocent victim. For example, the attacker may plant the victim's IP address in the logs of a web server which serves illegal content, a server which could be later captured by LEAs. After inspecting the logs of the captured server, LEAs may reasonably think that the victim has accessed illegal material. To make the attack even more convincing, the attacker may plant such bogus information on several different places which LEAs may check. In this way, the victim will be in a very difficult position when it comes to proving her innocence.

One potential solution to this problem would be to access the Internet through anonymizing services such as Tor and Freenet [DMS04, CSWH01]. Although such services protect a user's privacy, it is not clear to what extent they are effective against a malicious attack such as the one presented in this chapter. For example, if the attacker manages to learn the victim's IP address, he may still be able to plant bogus information on access logs. Thus, even if the victim tries to hide her Internet accesses, it may still be possible for an attacker to plant fabricated information leading straight to the IP address of the victim. Another potential solution to this problem would be to prohibit the collection of users' data. Although this approach could solve the problem, it is not likely that it is going to happen soon. For example, web site owners have been keeping, and continue to keep, access logs listing all the IP addresses of their customers along with the content they requested. It is not clear whether, or when, web site owners will cease to collect such information. Similarly, several organizations entice users to give their explicit consent in data collection by providing them with token benefits such as item discounts. To make matters worse, existing laws require several organizations, such as ISPs to collect and retain access data without requiring the user's consent. Thus, it is unclear, if not totally unlikely, that data collection about users' accesses on the Internet will cease in the near future. For this reason, we present a slightly different approach. Instead of adopting a system which hides or eliminates the network traces of a user, we propose a system which tracks and cryptographically proves (signs) all network activity, i.e. all IP addresses accessed by the user. If the user is falsely accused of committing an on-line crime, she will be able to prove that she has not accessed the IP addresses involved at the day of the crime. In addition, we provide a hashing scheme, which makes it practically impossible for someone to reveal all network history of a user, even if all the logs collected by our system are exposed in the wild. Unless the user gives her consent, revealing and proving that her traces are authentic is computationally very expensive.

8.1.1 Proposal

It seems that the root of the problem we study, is that several different organizations collect data which are not or can not, in hindsight, be double-checked for their correctness and authenticity. Incomplete, or fabricated, data, may in turn, be used to accuse users of illegal or inadmissible activities on the Internet. To rectify this problem, we empower users with the ability to record their own logs: the logs of their Internet accesses. ISPs, on behalf of the users, record every Internet access of the user. To make sure that the ISP does not record more than the user has accessed, users cryptographically sign every Internet access they make. ISPs, on their part, (i) keep a record of all signed accesses and (ii) honor all accesses for which they receive a signature. If the ISP receives a request to access a destination IP address, for which request no signature exists, the request does not go through (a `TCP-RST` packet is sent by the ISP for terminating the connection). We fully understand that keeping a log of all of a user's Internet accesses, even in an encrypted form, may potentially lead to a serious invasion of privacy if these accesses are leaked. To make sure that no such invasion can easily happen, the ISP does not keep the signed record itself, but only a hash of each signed record. Even if the hash table is leaked by an ISP it would be difficult to reconstruct the original signed record from the leaked hashes. In addition to this, an innovative use of salting which will be described later in this chapter makes this potential reconstruction extremely expensive even for determined attackers who resort to performing a dictionary attack.

We believe that our approach has several advantages:

1. If a user is falsely accused of accessing an IP address, a lookup in the logs can prove that no such signed record of this access can be found and, thus, no such access has been made.

2. It protects the users against malicious tampering with their log. All access records are signed by the user using her private key. As long as the private key is kept safe, it is computationally hard to insert fake records.

3. It is robust against leakage. Since only hashes of the signed records are stored, a potential leakage will provide only the hashes and not the original information.

4. It is resistant to dictionary attacks. The use of date in the calculation of hashed signed records along with a novel use of salting make dictionary attacks computationally expensive. This level of expensiveness can be fine-tuned by the user.

8.1.2 Contribution

The contributions of this chapter can be summarized as follows:

- We design, implement, and evaluate a framework that provides Network Flow Contracts (NFCs). An ISP providing NFCs can guarantee that all network activity has been logged with the subscriber's verification. More precisely, the ISP terminates all connections that have not been digital signed by the subscribers that generate them.

- We propose a lighter version of NFCs, specifically designed for web traffic through the addition of an HTTP header field, namely `X-Signature`. An ISP providing NFCs for the web can guarantee that all web sessions of each subscriber have been logged with her consent.

- We design, implement and evaluate Dig Encryption, a framework for providing querying in encrypted data, which is able to resist dictionary attacks.

The rest of this chapter is organized as follows. In Section 8.2 we discuss in detail the threat models we are addressing. In Section 8.3 we present the overview of an architecture for providing NFCs, we describe how NFCs can be simply enabled for web traffic through the addition of an HTTP header and we give a high-level description of Dig Encryption. In Section 8.4 we perform a performance evaluation for NFCs and Dig Encryption. We discuss ideas for future work in Section 8.5.

8.2 Threat Model

In this section we present in detail the threat models we address with NFCs.

8.2.1 Overview

Digital data has become an important factor that characterizes the every-day activities of our society. Activities performed in the network, such as serving or *accessing* resources with pedophiliac content, can be associated with criminal or improper behavior. In certain cases, such activities can be enough for filing a lawsuit against an Internet end-user. It is hard to find official evidence about cases where people have been convicted for crime, because they simply accessed a web site. However, there are numerous anecdotal cases describing a trend for adopting certain measures and technologies for monitoring the user activity in the network.

For example, the recent case of WikiLeaks inspired many reporters to author stories about systems employed from national agencies, in order to track users that access the content provided by the numerous WikiLeaks mirrors all over the world. Some news stories provided technical details about HoneyPots that can attract and record users who are interested in the WikiLeaks content [cia]. In a similar but more officially positioned act, the European Union (EU) has expressed interest in monitoring all queries submitted to search engines [eu-], in order to reduce activities connected to pedophiles.

Moreover, each country maintains specific laws and procedures for data retention associated with data collected by the telecommunications providers. For example, EU suggests that all data must be kept for a period of 6 months to 2 years [dat]. In the United States (US) there are similar laws and, recently, in the context of the Internet Stopping Adults Facilitating the Exploitation of Today's Youth (SAFETY) Act a bill was published [saf] enumerating specifically which digital data is to be stored for further tracking of a suspect.

We address concerns about possible implications and privacy issues related to data retention. We consider that in a constantly evolving world of digital media, the data collection can become arbitrary and eventually control will be lost. On the other hand, the user has limited control on data collection process and has little means to defend against accusations that stem from fake or false data. We do not attempt to *alter* existing methods for data collection or propose a new data retention scheme. Instead, we aim at developing novel technologies that can shield innocent users from groundless claims, emerging from digital data. These, technologies can run in parallel with existing schemes.

We now proceed and list some possible scenarios where our proposal can be applicable.

8.2.2 Malicious Web Sites

The easiest way to create fake evidence of a user's network activity is for a web site to create artificial logs. This can happen in numerous occasions. First, the operator and owner of the web site can manipulate the data contained in the log file. Second, an insider employee, having access to the log infrastructure, can alter the information. Consider that many popular web sites occupy thousands of employees and many of them have the opportunity to leak (or modify) private data [Tec]. Finally, a bug in the web site's code can be exploited by an external attacker or by malware and give the opportunity to a third party to inject fake records in the web site's logs.

In all cases, data must be correlated with information owned by other providers, such as ISPs or cell phone providers. However, it is possible for the correlation procedure to have negative results, because a cell provider, for example, has lost or delete the records. Eventually, the web site's fake logs will be the only evidence in this particular case. Thus, an unsuspecting user can be incriminated and have limited flexibility in providing further evidence in order to defend herself.

8.2.3 Untrusted ISPs

Nowadays, there is no proven trust between an ISP and its subscribers. This claim can be extended for other digital media providers, such as telephony providers, but we limit the scope of this work to ISPs. The end-user routes the traffic through the ISP, but has limited control on the data collected in regards to her network activity. An untrusted ISP can easily create fake evidence of the user's network accesses. To make things worse, the ISP is considered the only authoritative source of information that can prove that a particular subscriber performed a series of network actions. In the case of a rogue ISP, which provides fake log records, the user can be seriously incriminated and unable to defend herself.

We consider the probability of rogue ISPs significant, since the number of ISPs world-

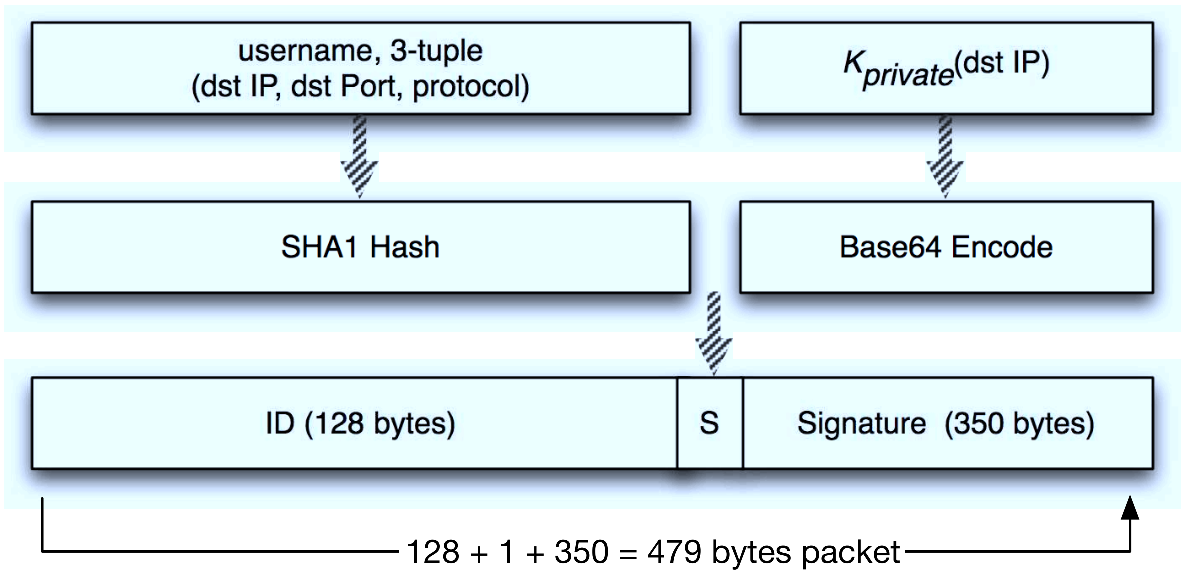


Figure 8.1: NFC packet format. The packet is composed by an identifier, ID, which characterizes the connection, a delimiter S and the Base64 encoding of the signature. The ID is formed using the SHA1 key generated by hashing the subscriber name concatenated with the 3-tuple information and the signature is the destination IP encrypted with the user's private key.

wide is constantly growing. To this end, there are numerous micro-ISPs that purchase Internet in bulk from providers owning a backbone and offer it to subscribers at appealing prices. Cases where micro-ISPs were affiliated with illegal activities have been recorded in the past [DMPW09]. In the majority, these activities refer to SPAM and malware distribution. However, it is still questionable if ISPs will take advantage of the data they collect in the future.

8.3 Architecture

In this section we provide a detailed description of our proposal, Network Flow Contracts (NFCs); an architecture for ensuring provable agreement between an ISP and a subscriber for any network access. We propose that each network flow is signed by the user, with the assistance of public-key cryptography. In this section we analyze the complete architecture for signing network flows and producing NFCs and we also present a case study involving web traffic. Finally, we describe *Dig Encryption*, a privacy-aware encryption scheme for

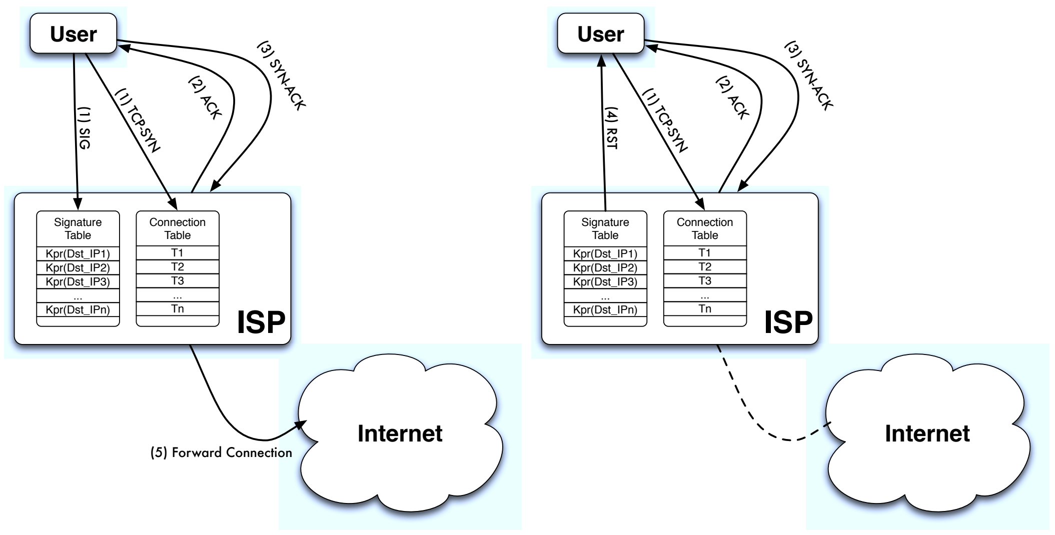


Figure 8.2: The NFC architecture. The ISP maintains for each on-line subscriber two tables: (a) the Connection Table, and (b) the Signature Table. Every new flow is stored in the Connection Table and incoming signatures are stored in the Signature Table. If, after time T, no signature for a new flow is present in the Signature Table the new flow is terminated.

storing NFCs. Dig encryption provides limited data querying in a data-set by imposing time constraints and, thus, it is resistant to brute force attacks.

8.3.1 Overview

NFCs must be offered by ISPs as a service to users willing to protect themselves from incrimination attacks carried out using artificially modified data logs. NFCs are not a new data retention scheme and do not aim on substituting existing data retention schemes. On the contrary, NFCs is a new service and users that care to provably protect their network actions can take advantage of it.

NFCs must be deployed in the ISP and they need additional software running in the user's host. The client-side software must not be distributed by the ISP, since we assume that in the general case the ISP is not trusted. We present a generic implementation of both the server and client part of the software. In a real deployment, the client part can be implemented in the kernel, by OS developers, or in user space as a standalone application.

In a nutshell, the enforcement of NFCs works as follows:

1. The subscriber registers for the service using a web portal operated by the ISP by

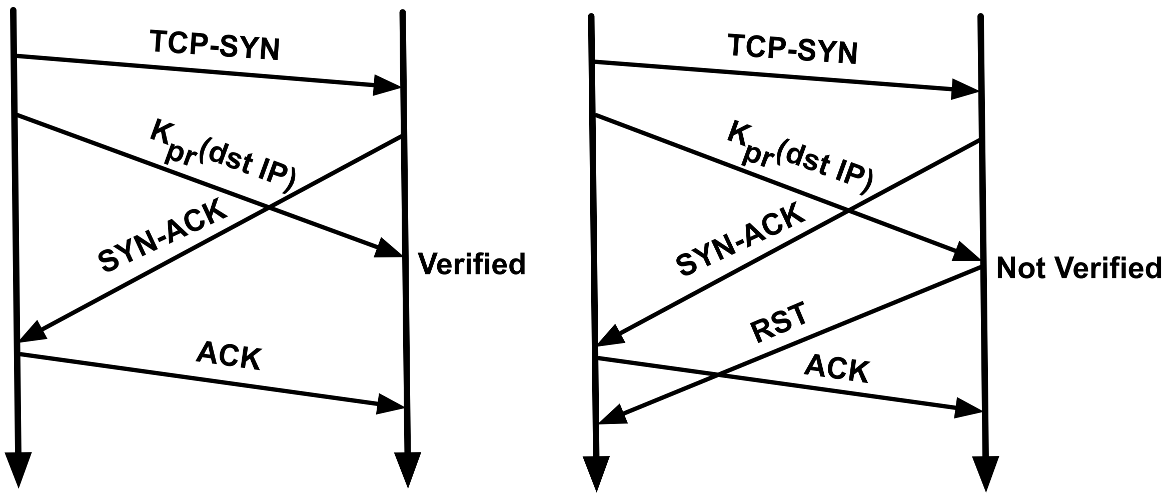


Figure 8.3: The NFC protocol. For every incoming `TCP-SYN` the ISP waits for the NFC that signs the new flow. In the case that no signature is received after T seconds from the time the original `TCP-SYN` was observed, the ISP terminates the subscriber's connection by sending an explicit `TCP-RST`.

uploading her public key and presumably paying an indicative fee.

2. The subscriber installs a software client, if the OS does not support NFCs, which is able to sign all outgoing traffic routed by the ISP.

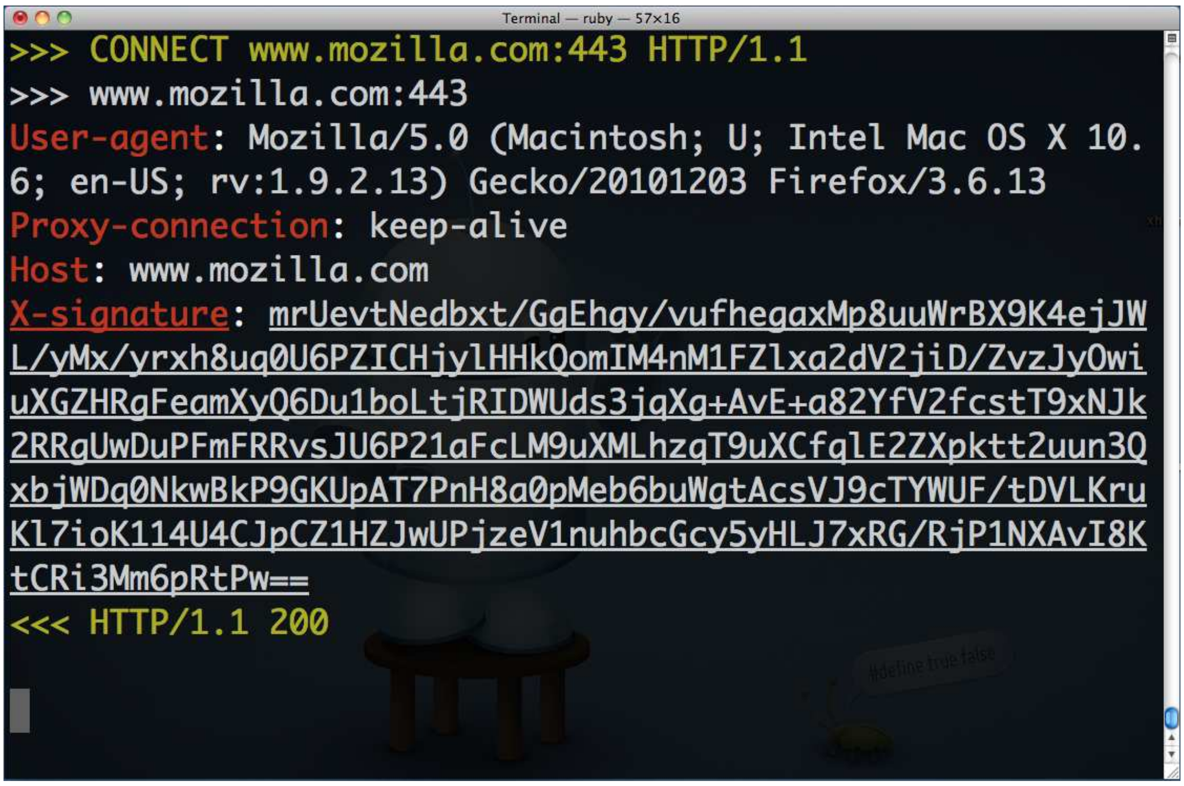
3. For every new flow created by the subscriber, the ISP waits for time T to receive the flows's signature. If the signature is not received, the ISP terminates the subscriber's connection by explicitly sending a `TCP-RST`.

This scheme is applicable to all TCP traffic generated by the subscriber. We consider only TCP traffic, since usually this type of traffic is connected with state-full network activities. Services that communicate over UDP always have a communication part over TCP in order to authenticate the user. In Section 8.5 we discuss possible ways to expand NFCs for UDP traffic.

Below we analyze in more depth, all the internals of NFCs.

8.3.2 Network Flow Contracts

An NFC is a digital contract between a user and the ISP. For all further discussion, we assume a DSL user connected to an ISP, but the scheme is quite generic to be applicable in other



```
>>> CONNECT www.mozilla.com:443 HTTP/1.1
>>> www.mozilla.com:443
User-agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.
6; en-US; rv:1.9.2.13) Gecko/20101203 Firefox/3.6.13
Proxy-connection: keep-alive
Host: www.mozilla.com
X-signature: mrUevtNedbxt/GgEhgy/vufhegaxMp8uuWrBX9K4ejJW
L/yMx/yrxh8uq0U6PZICHjylHHkQomIM4nM1FZlxa2dV2jiD/ZvzJyOwi
uXGZHRgFeamXyQ6Du1boLtjRIDWUds3jqXg+AvE+a82YfV2fcstT9xNJk
2RRgUwDuPFmFRRvsJU6P21aFcLM9uXMLhzqT9uXCfqlE2ZXpktt2uun3Q
xbjWDq0NkwBkP9GKUpAT7PnH8a0pMeb6buWgtAcsVJ9cTYWUF/tDVLKru
Kl7ioK114U4CJpCZ1HZJwUPjzeV1nuhbcGcy5yHLJ7xRG/RjP1NXAvI8K
tCRi3Mm6pRtPw==
<<< HTTP/1.1 200
```

Figure 8.4: Enabling NFCs for web traffic requires the addition of a new HTTP header, which simply signs with the user's private key the URL resource that the browser is requesting.

cases, such as users connected to a telephony network. For generating an NFC the subscriber is required to own a pair of public and private key, and operate an NFC client. For running the NFC client, the user must provide the software with her private key. The NFC client passively monitors all traffic generated by the subscriber's host. For each outgoing `TCP-SYN` packet, the client retrieves the destination IP address of the packet, it signs the IP address with her private key and sends the packet containing the signature to an NFC server operated by the ISP. Digital signatures are usually associated with large documents, where the term digital signature refers to the outcome of encrypting the document's hash with a private key [Sch07]. In this work, we sign IP addresses and thus we do not use any cryptographic hash function in the signing process.

The ISP maintains for each on-line subscriber two tables: (a) the Connection Table, and (b) the Signature Table.

Connection Table. The ISP generates a record for each incoming `TCP-SYN` packet received by the subscriber's host. The record is constructed by taking the user's identity and the key produced by hashing the 3-tuple (destination IP, destination port and protocol) of the new TCP connection. We do not rely on the 5-tuple (source IP/port, destination IP/port, protocol), because usually NAT/Proxies can overwrite the source part [Cla]. The ISP proceeds and routes the `TCP-SYN` packet to the destination, and in parallel monitors the Signature Table for a signature associated with the new TCP connection. In the case that no signature has been found in the Signature Table after T seconds from the time the original `TCP-SYN` has been observed, the ISP terminates the subscriber's connection by sending an explicit `TCP-RST`. Later, in Section 8.4 we estimate appropriate values for T.

Signature Table. The Signature Table is maintained by an NFC server. A user, who has been registered to the service, is using an NFC client, which is constantly connected with the ISP's NFC server. This connection serves as the information channel that communicates all NFCs (i.e. the signatures) from the subscriber's host to the ISP. For each new outgoing TCP connection, the NFC client sends a packet like the one depicted in Figure 8.1. The packet is composed by an identifier, ID, which characterizes the connection, a delimiter S and the Base64 encoding of the signature. The ID is formed using the SHA1 key generated by hashing the subscriber's name concatenated with the 3-tuple information and the signature is the destination IP encrypted with the user's private key.

An overview of the NFC architecture is depicted in Figure 8.2. The NFC protocol is presented, step-by-step, in Figure 8.3.

Provable Network Activity. With the assistance of NFCs a user can prove all network activities issued through the ISP, which supports the service. Consider the case where Alice is facing possible claims accusing her of having performed illegal network activities. Such activities include the visiting of web sites serving illegal content, participating in torrents exchanging copyrighted material and any other network activity that is subject to current law. We assume that Alice did not perform any of these network accesses. We now review the threat models presented in Section 8.2 and discuss how NFCs can protect Alice from all faulty claims.

- *Incrimination driven by a web site.* A web site can provide LEAs with fake logs that indicate that Alice has been one of its visitors in the past. Although, logs produced by a web server are hard to stand as evidence for an illegal activity, they can be combined with further information and, thus, have impact on LEAs' decision. In the case that NFCs are used, the web site must also provide further information, such as the IP address of the machine Alice used to access the web site. Moreover, the IP address will be looked-up in the ISP's data logs in order for the corresponding signature that identifies Alice's connection to be located. If no signature is found, then the logs of the web site cannot be used for accusing Alice of any access.

- *Incrimination driven by an ISP.* A more powerful threat model, as described in Section 8.2 is when a web server, or any other Internet service, is colluding with the ISP in order to incriminate Alice and accuse her for accessing illegal resources. In this case, the evidence is more solid. Alice is accused of accessing an illegal resource, using fabricated logs produced both by the illegal resource and the ISP which routes the user's network traffic. However, if Alice has registered with the NFC service, the ISP must provide the IP address of the destination resource signed with Alice's private key. Assuming that a modern public encryption algorithm [RSA83] is used and the private key of Alice is kept secret, then producing a fake digital signature of the destination IP address is considered computationally very hard.

8.3.3 Web Traffic

NFCs can be used for creating contracts between a user and an ISP for any network activity. Nevertheless, a lighter scheme can be applied only for web traffic. We consider, that web accessing can be associated more easily with illegal activities performed by the average end user. Applying NFCs for web traffic is easier, as far as implementation is concerned, and it is characterized by easy deployment, especially on the client-side.

Enabling NFCs for web traffic requires the addition of a new HTTP header, which simply signs the URL resource that the browser is requesting with the user's private key. The majority of modern web browsers support extensions that can modify the HTTP headers of every HTTP request performed by the browser [mod]. Instead of creating web browser extensions, we choose to create a generic HTTP proxy, which runs on the client machine and adds the proposed `X-Signature` header field to the HTTP headers of every outgoing request. We configure the custom web proxy to attach the special `X-Signature` field to every outgoing HTTP request in the following way. The proxy receives the HTTP request, extracts the target URL, encrypts the URL with the user's private key, encodes the result in Base64, because of the fact that HTTP is a text-only protocol, and, finally, attaches the `X-Signature` field, which has as a value the produced signature. The request is then forwarded to its destination. In Figure 8.4 we depict a screenshot showing the custom proxy in action.

Again, the HTTP logs can be altered or artificial HTTP logs can be constructed. However, the attacker has to also inject fake `X-Signature` fields, that carry out the destination URL signed with the user's private key. One crucial parameter in web NFCs is that the ISP has to perform deep packet inspection in order to extract the `X-Signature` value and verify the signature. We further discuss this issue in Section 8.4.

8.3.4 Dig Encryption

NFCs require the ISP to keep all data logs associated with each user's network activity. The data logs must contain all hosts that came in contact with the ISP's subscribers, as well as the signatures generated for each connection. There are specific laws for digital data maintenance in telecommunications. In Europe, for example, the provider must keep all data logs produced by the costumers for a period of 6 months to 2 years [dat]. Similar laws apply in the US. This strategy can assist in tracing crimes associated with network activity. However, keeping all data for large period of times is risky, as far as the privacy of the subscribers is concerned, since the data can be leaked [Tec] or stolen by a third party. An option for dealing with data leakage and theft is the ISP to keep all data encrypted. Researchers have designed and implemented practical techniques for querying encrypted data [SWP00]. Thus, an ISP could have all data encrypted and partially provide answers to specific queries only in the context of a trial or when a warrant is issued. Unfortunately, in the case of NFCs, encrypting the data is not efficient. An ISP that stores all data accesses and all signatures encrypted is still vulnerable to a brute force attack. Consider the case, where LEAs ask the ISP if a particular subscriber accessed *all* existing URLs. This set of queries will reveal the user's complete network activity. In the case where the user is not guilty, this process will result in the user loosing all privacy. Thus, we optimally seek for a method that will allow LEAs to query an ISP for the network activity of a particular subscriber, but without being able to perform an arbitrary number of queries. We also seek a method that prevents anybody from exposing a user's privacy, even if the ISP's facilities are compromised and third parties grant access to the log files.

In this work, we propose *Dig Encryption*, or simply *Digging*, which is a method for storing only the necessary information in order to provide time-constrained querying. We now describe an overview of all the steps an ISP must follow, in order to provide dig encryption:

1. The ISP receives an incoming `TCP-SYN` packet and a signature for this new connection. The signature has been produced by encrypting the destination IP address of this new connection using the user's private key.

2. The ISP proceeds to decrypt the signature with the user's public key in order to verify it. If the result is indeed the destination IP address of this new connection the ISP keeps the signature. Otherwise it transmits a `TCP-RST` to the user's host, in order to terminate the connection.

3. Assuming that the signature is authentic, the ISP produces the Base64 encoding of the signature. It then concatenates the current date and a random number, used as a salt and taken from a range of length R, to the Base64 representation of the signature. It finally, produces a cryptographic hash by hashing the result of the concatenation. The ISP deletes all information, including the salt, and keeps only the final cryptographic hash.

The above is an overview of the whole process. In Section 8.4 we provide more details for the implementation, the cryptographic algorithms used, as well as how large R is.

Now, we assume that there is a case under investigation and there is a warrant issued, that allows LEAs to query an ISP for the network profile of the suspect. Using the above scheme, the ISP by design is able to answer only if a particular subscriber has been connected with a particular host on a specific date. For example, LEAs can form a query which is composed by the user, Alice, an IP address, Bob's machine, and a date D. This query is compiled in: *Did Alice connect to Bob's machine on date D?* In order for the ISP to answer this question the procedure below is followed:

1. Alice, the suspect, encrypts Bob's IP address with her private key and submits it to the LEAs.

2. The LEAs verify that the produced signature is authentic by decrypting the result, which Alice submitted, with Alice's public key. If the product is indeed Bob's IP address, then the signature is considered authentic. If the signature is authentic, the LEAs submit the signature and the date D to the ISP for further processing.

3. The ISP takes the signature, it produces the Base64 encoding and concatenates the result with the date D. It then proceeds and generates the products of the concatenation of the final result with all possible numbers up to R.

4. The ISP produces all cryptographic hashes by hashing all concatenations.

5. The ISP checks to see if any of the produced hashes exists in the storage, which contains all hashes associated with Alice's network activity and replies to LEAs positively or negatively according to the result.

The above procedure is required for answering one query. The time needed to complete the whole procedure depends on the chosen value of R and the algorithms used for the cryptographic hashing. In Section 8.4 we discuss all these properties and we suggest values for imposing acceptable constraints. Observe, that for each query the ISP has to *dig* in the storage, by performing a classic brute force attack in the salt's space, in order to resolve if the signature has been recorded or not.

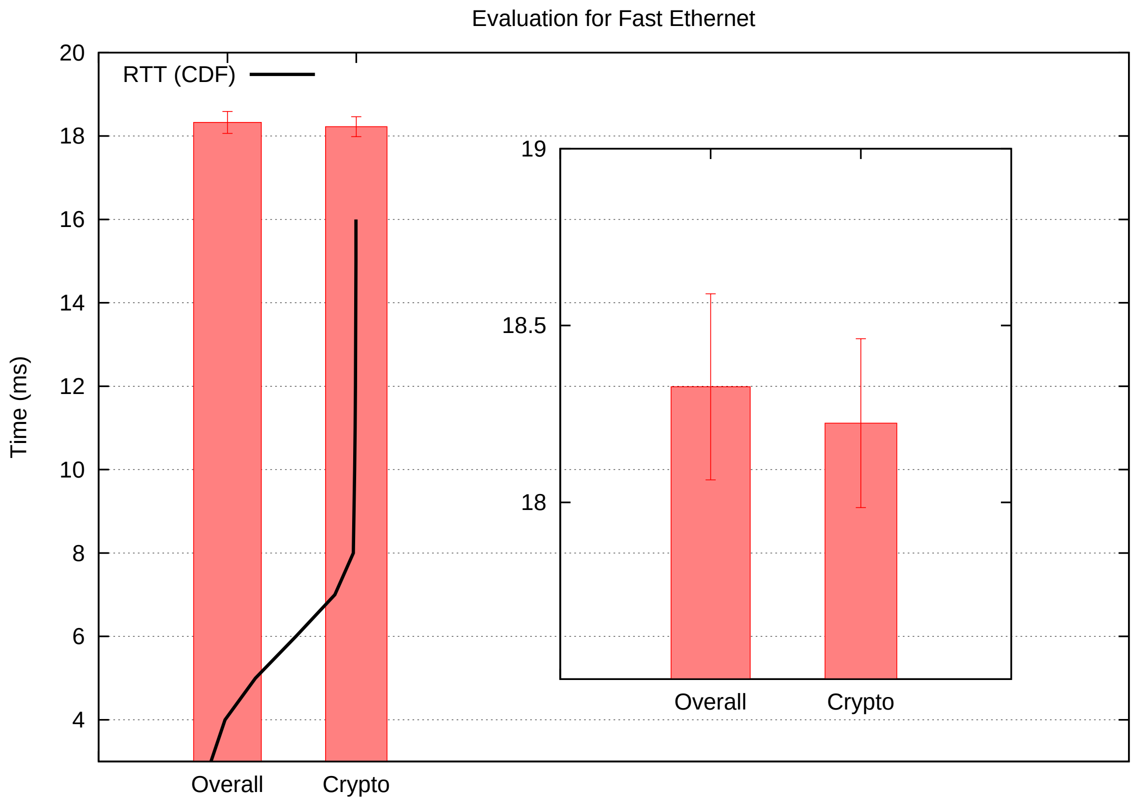


Figure 8.5: NFCs evaluation for an NFC client connected with an NFC server through a Fast Ethernet network.

8.4 Evaluation

In this section we provide technical details about our own implementation of both NFCs and dig encryption. We further proceed and evaluate our implementations and propose a configuration applicable to any ISP that wishes to enable NFCs as a service.

8.4.1 NFCs

We implement NFCs in Linux using `libpcap` [JLM94] as a basic building block for capturing network traffic, and OpenSSL [VMC02] for all cryptographic operations. All software is written in C++ and does not exceed 2,000 LoCs. The setup is the following. A Linux-based host, emulating Alice's host, runs the `signer` software. The software initially connects to a predefined open port of a Linux based server, which emulates the ISP. It then proceeds and monitors all TCP traffic for the `TCP-SYN` flag. For each captured packet, `signer` generates a

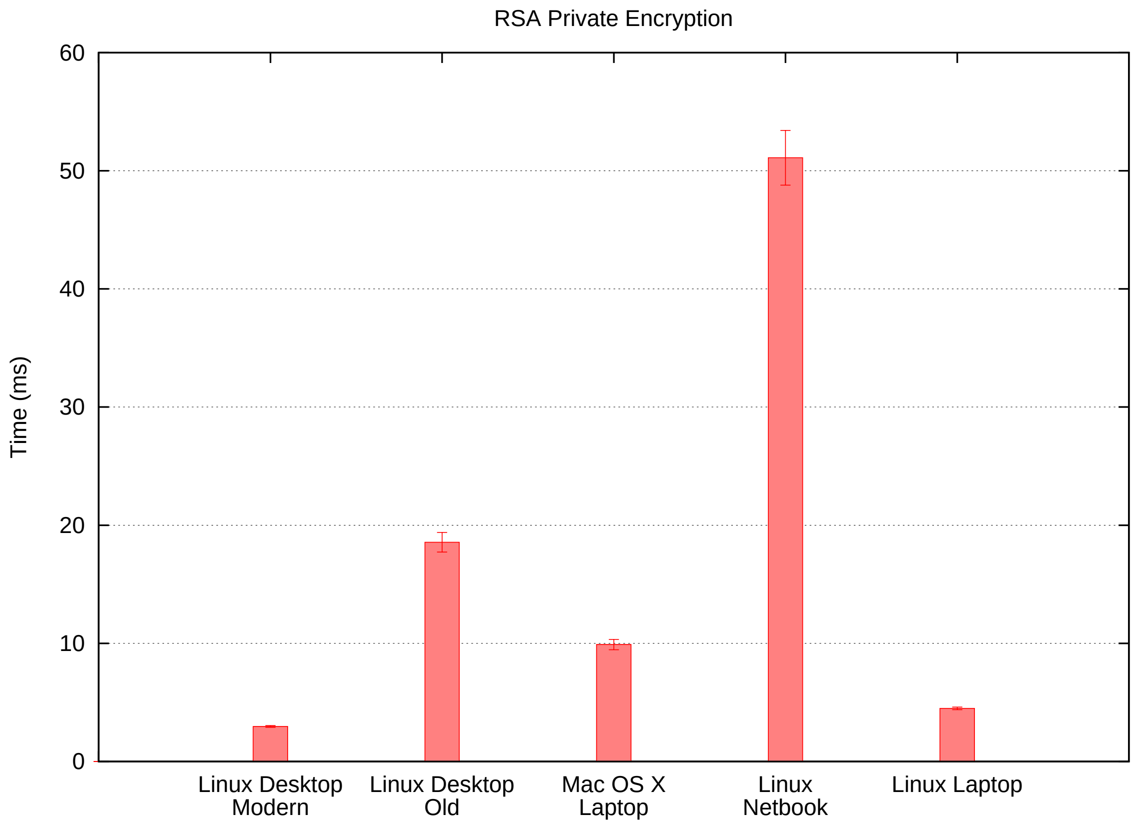
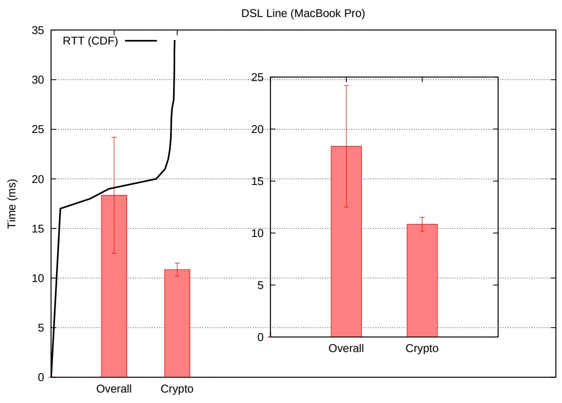


Figure 8.6: Overhead of RSA encryption with a private key for different architectures.

signature by encrypting the destination IP of the packet with Alice's private key and sends it to the server using the established connection. The server, also, runs custom software, the `signer-server`, in order to capture all incoming `TCP` packets that have the `TCP-SYN` flag on, extract the 3-tuple information and store it in a hash table. In addition, `signer-server` listens to a predefined port for incoming signatures. The packet that contains a signature is depicted in Figure 8.1. For each incoming signature, the server verifies it by decrypting it using the user's public key. If it is authentic it stores it in a hash table, using dig encryption (see later in this section for the technical details of dig encryption). If, for any reason, the signature is not authentic or the signature arrives after time T the connection is terminated by sending an explicit `TCP-RST` to Alice's host. In this section we estimate and suggest typical values for T. For the public-key encryption/decryption RSA is used (the `RSA_PKCS1_PADDING` algorithm provided by OpenSSL). The length of the keys is 2048 bits. For the hash tables, the data structures provided by STL [MS95] are used.



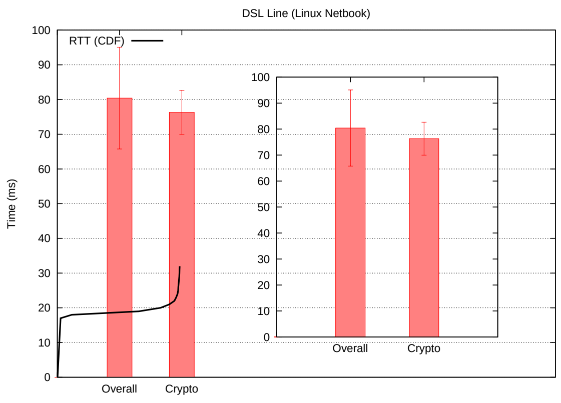


Figure 8.7: NFCs evaluation for an NFC client connected with an NFC server through a DSL line.

In order to estimate typical values of T we proceed and carry out the following experiment. A client running the `signer` software is connected to a server, running `signer-server`, as described above. The client generates 100,000 `TCP-SYN` requests towards the server, using the `hping3` tool in fast mode (10 requests per second). We record the followings: (a) the time needed for the client to create a signature (this is the *Crypto* part), (b) the time elapsed from the arrival of the `TCP-SYN` packet to the arrival of the signature, as recorded by the server (this is the *Overall* part), and (c) the RTT as reported by `hping3`. We conduct the experiment for two configurations. The first configuration is composed by two Linux desktops connected through a Fast Ethernet (FE) network. The client machine is a 6 years old Linux box and the server machine is a modern Linux box with state-of-the-art hardware. The second setup is composed by a MacBook Pro laptop and a Linux netbook connected with the Linux server, through a DSL line. We choose these configurations for the following reasons. As far as the FE experiment is concerned, where network overheads are negligible, we select the client to be an old PC with moderate computational capabilities, since we seek to focus on the overhead imposed by the cryptographic operations. As far as the DSL setup is concerned, we select a typical commodity laptop (MacBook Pro), representing the average user, and a Linux netbook, with low computational resources, representing a mobile user.

In Figure 8.5 we depict the results for the FE case. More precisely, we plot a histogram for the average inter-arrival time of the `TCP-SYN` and the signature, and the average time spent in cryptographic operations. We also plot the CDF of RTTs; the distance between the

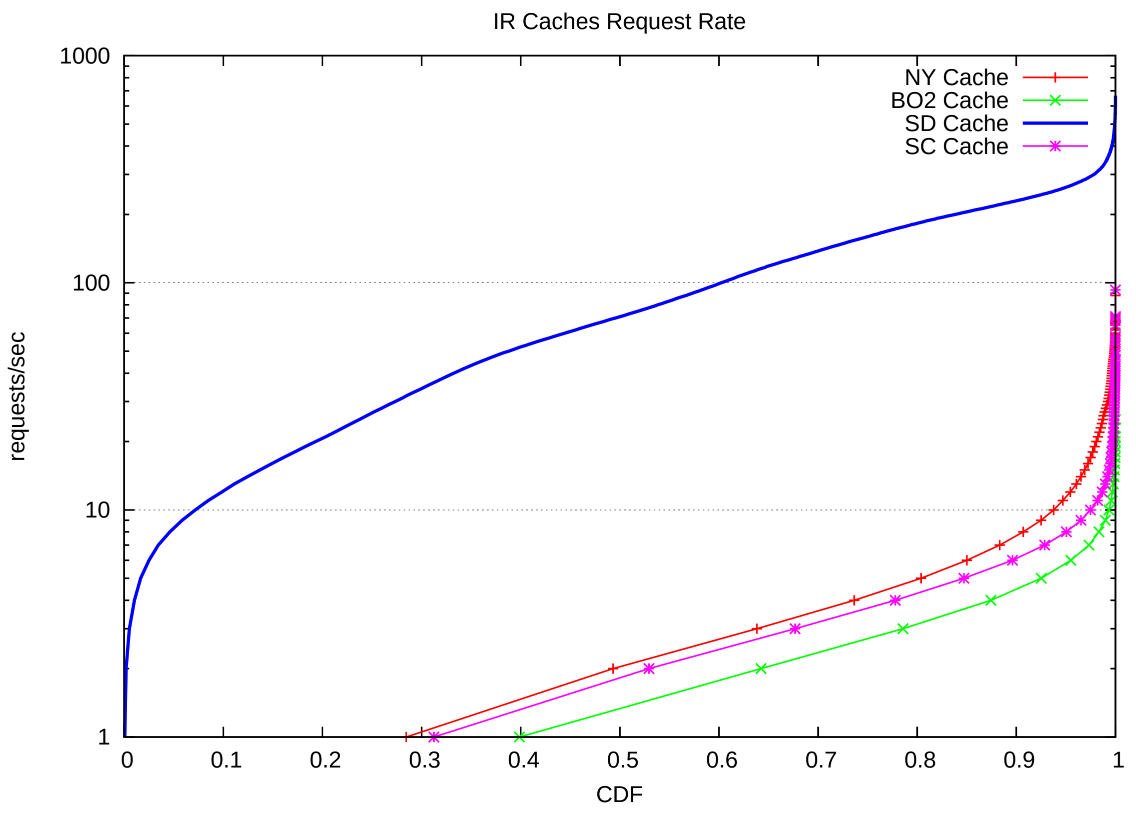


Figure 8.8: CDF of the request rate extracted from four web caches obtained by IRCache [irc] in the period of one day. Notice, that the average request rate for the heavy loaded cache is a few hundred requests/sec.

beginning of the positive part of x-axis and the *Crypto* bar is 1. A first look suggests that the signature arrival time is greater than a typical RTT value. This is notably because of the computational part associated with cryptographic operations. To support this claim, in Figure 8.5 we also plot a second graph zooming in the top of the histogram. Notice, that the majority of the time is spent on cryptographic operations, and the network lag, which involves the inter-arrival of the `TCP-SYN` and the signatures packet at the server is in the scale of the RTT as reported by `hping3`. Another interesting remark is the following. Notice that if the standard deviation (depicted in the plot with the vertical error bar) associated with the crypto computation is accounted, the crypto part can exceed the overall time required for the two network packets, namely the `TCP-SYN` and the signature, to arrive at the server. This is due to the fact that there is buffering in the network driver of the client causing the `TCP-SYN` packet to actually leave the host after the cryptographic computation has already started.

```ruby
def dig(message, r)
  salt = rand(r)
  signature =
  Base64.encode64(@user_private_key.private_encrypt(message))
  c = message + salt.to_s + Date.now()
  hv = OpenSSL::Digest::Digest.new("sha512").hexdigest(c)
  return(hv)
end

def undig(message, r, d, hv)
  signature =
  Base64.encode64(@user_private_key.private_encrypt(message))
  r.times do |i|
    c = message + i.to_s + d
    attempt =
    OpenSSL::Digest::Digest.new("sha512").hexdigest(c)
    if attempt == hv
      puts "Message is found at attempt: #{i}."
      break
    end
  end
end
```

Figure 8.9: Prototype implementation of dig encryption in Ruby.

For a better estimation of the cryptographic part, we perform 10,000 private-key encryptions using the RSA implementation provided by OpenSSL in five different architectures, namely: (a) a modern Linux desktop with state-of-the art hardware (Intel i7, 4 GB RAM), (b) a 6-year old modest Linux desktop; the one used in the evaluation over FE, (c) a MacBook Pro running Mac OS X, (d) a Linux netbook and (e) a modern Linux laptop. We provide the results in Figure 8.6. Notice, that the computational overhead in all platforms, apart from the old Linux desktop and the Linux netbook is less than 10 ms, which is of the same order of magnitude with a typical RTT of a FE network.

We repeat the initial experiment on a DSL line. In a similar fashion, we depict the results, using the same conventions, in Figure 8.7. Notice, that in the case of the MacBook Pro laptop, the inter-arrival time between the `TCP-SYN` and the signature is compared to the average RTT between the laptop and the server. The cryptographic part takes nearly 10 ms, which is approximately equal to the one-way delay. In the case of the Linux netbook, the

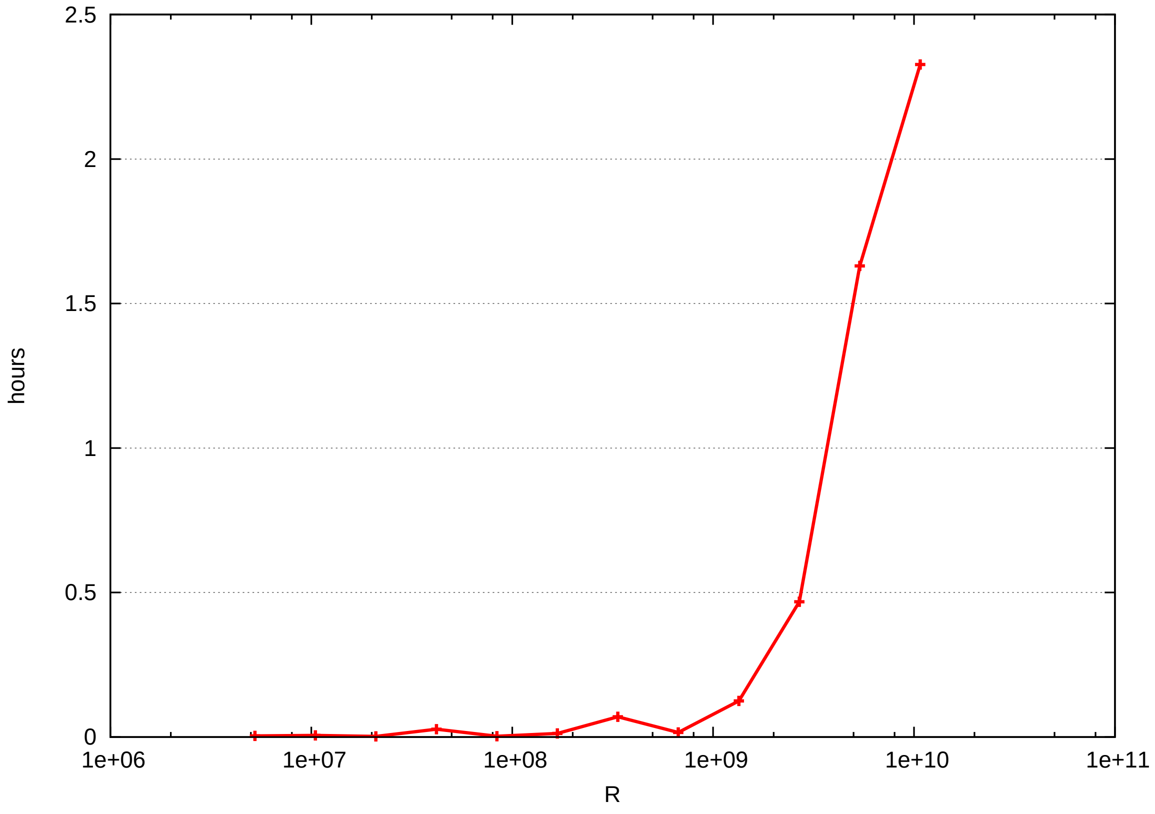


Figure 8.10: The time which is taken to *undig* a message for various ranges of salt's space in hours.

cryptographic operation dominates, but again the overall delay is less than 100 ms which is an acceptable delay. Recall, that this overhead is experienced only during the establishment of a new TCP connection.

Thus, we conclude that NFCs impose an overhead due to cryptographic operations only when low-computational devices are used. The overhead is not dramatic and it is in the order of magnitude of an RTT, as far as broadband networks are concerned. A typical safe value of T can be 100 ms. The software run by the ISP can perform some initial tests in order to benchmark the computational capabilities in cryptographic operations of the client's device for selecting a more accurate threshold.

8.4.2 Server Overhead

For each incoming signature the NFC server should verify the signature by performing an RSA decryption using a public key. The cost of decryption is proportional to encryption using a private key in RSA (see Figure 8.6 for some typical values in different architectures). The overall overhead of the verification process depends of the amount of traffic the ISP has to route each time. We use data obtained by IRCache [irc] for approximating typical volumes of web traffic. The IRCache project provides anonymized web traces collected by a set of web caches running the Squid software. In Figure 8.8 the CDF of the request rate extracted from four web caches in the period of one day is plotted. Notice, that the average request rate is for the heavy loaded cache a few hundred requests/sec. Assuming that the NFC server can verify a signature in milliseconds, then a typical verification for web traffic imposes an overhead of seconds. The overhead can be reduced by employing custom hardware acceleration. Consider that Sun's UltraSPARC T1, equipped with a Modular Arithmetic Unit for RSA, could perform 20,425 signature verifications per second using a 2048-bit key utilizing all 32 cores, 6 years ago [t1].

8.4.3 Dig Encryption

We provide a prototype implementation of dig encryption in Ruby. Part of the code is depicted in Figure 8.9, where we list the `dig()` and `undig()` functions. We now describe these implementations in detail.

The `dig()` function takes a message as input and an R value, which is a suggested range for the random salt. The function encrypts the message with the user's private key using the RSA algorithm and it computes the Base64 encoding of the encryption's result. It then adds a random value, selected from a range of length R, and the current date to the Base64 representation of the encryption and it computes the SHA512 digest. It finally returns the last product, which is the cryptographic hash.

The `undig()` function takes a message as input, a suggested R value, a date d and a cryptographic hash hv. The purpose of the `undig()` function is to find a cryptographic hash, which matches hv. The function encrypts the message with the user's private key using the

RSA algorithm and it computes the Base64 encoding of the encryption's result. It then loops all possible values, selected from a range of length R. For each value, it concatenates it along with the current date to the Base64 representation of the encryption and it computes the SHA512 digest. If the digest matches hv it reports that the `undig` operation terminated successfully.

In Figure 8.10 we plot the time taken for an `undig` operation for various values of R. Notice that for large values of R, un-digging an encrypted signature can take hours.

8.5 Discussion and Future Work

In this section we discuss some design issues we choose in more detail. We also present our plans for future work.

8.5.1 Traceability

In Section 8.3.4 we presented an encryption scheme for storing NFCs as being logged by the ISP. Dig encryption requires the user's consent, specifically her private key, for investigating if a particular network access has taken place. An ISP that stores NFCs using dig encryption and does not perform any other form of data retention makes traceability of a user hard. For example, consider that Bob is registering for the NFC service. He later performs illegal activities and finally he deletes his private key, in order to eliminate all traces related to his activity.

NFCs and dig encryption do not aim on substituting any data retention scheme used by current ISPs, that allow further investigation about a user's network history. Instead, NFCs and dig encryption guarantee that a user can provide provable evidence that she has not accessed a network resource, if this has actually never happened. A user that lost or deleted on purpose her private key cannot take advantage of our system to prove anything about her network history. Instead, the user is subject to current laws and tactics applied in data logs provided by ISPs.

8.5.2 UDP Traffic

NFCs are applied to TCP traffic. The vast majority of Internet applications use TCP [BHH+10]. However, a user can also generate UDP traffic, which in turn can be logged by the ISP. UDP has no state and thus a UDP packet can easily be spoofed. Thus, it is hard to even claim that UDP traffic can provide any evidence of two hosts communicating over the Internet. On the other hand, services communicate partially over UDP; all processes that identify a user communicate over TCP. For example, VoIP applications use TCP for establishing connections and UDP for delivering audio/video. NFCs could be applied in a sampled portion of UDP traffic, but it is still questionable whether this could be practical.

8.5.3 RSA Cost

Traditionally, public-key cryptography is computationally expensive. Throughout this chapter, we have presented various measurements regarding the computational overhead imposed on signing and verifying a signature. As far as the client is concerned, we argue that RSA encryption does not impose any dramatic overhead that can alter the user's perception. As we showed in Section 8.4, even a low-power netbook needs less than 100 ms to compute a signature with a 2048-bit key. RSA encryption can be rather expensive for mobile devices, such as smart-phones. We leave this for future work. As far as the server is concerned, hardware acceleration can be of great assistance. Consider that a Sun's UltraSPARC T1, equipped with a Modular Arithmetic Unit for RSA, could perform 20,425 signature verifications per second using a 2048-bit key utilizing all 32 cores, 6 years ago [t1].

Chapter 9

Related Work

In this chapter we present related to the content of this dissertation research efforts.

9.1 Web Exploitation and Defenses

The closest studies to xJS and RaJa are BEEP [JSH07], Noncespaces [GC09] and DSI [NSS09]. We have highlighted certain cases where the aforementioned methodologies fail (e.g., see Chapter 3). We have presented attacks that escape whitelisting (proposed in [JSH07]) and cases where DOM-based solutions [GC09, NSS09] are not efficient. Our frameworks can cope with XSS attacks that escape whitelisting (attacks presented in Chapter 3), and do not require any information related to DOM; both xJS and RaJa can also prevent attacks that leverage the content-sniffing algorithms of web browsers [BCS09].

Blueprint [TLV09] is a server-only approach which guarantees that untrusted content is not executed. The application server pre-renders the page and serves each web document in a form in which all dynamic content is correctly escaped to avoid possible code injections. However, Blueprint requires the web programmer to inject possible unsafe content (for example comments of a blog story) using a specific Blueprint API in PHP. Spotting all unsafe code fragments of a web application is not trivial. Blueprint further imposes a significant overhead compared to solutions based on natively browser modifications, like xJS. On the other hand, as we have shown in Chapter 5, real-world applications, with minimal code-modifications, can utilize RaJa, without employing a custom API.

Enforcing separation between structure and content is another prevention scheme for code injections [RV09]. This proposed framework can deal with XSS attacks as well as SQL injections. As far as XSS is concerned, the basic idea is that each web document has a well defined structure in contrast to a stream of bytes, as it is served nowadays by web servers. This allows the authors to enforce a separation between the authentic document's structure and the untrusted dynamic content from user input, which is attached to it. However, in contrast to xJS and RaJa, this technique cannot deal with attacks that are based on the content-sniffing algorithms of browsers [BCS09] as well as attacks that modify the DOM structure using purely client-side code [Kle].

9.1.1 Data Tainting

In [VNJ$^+$07] the authors propose to use dynamic tainting analysis to prevent XSS attacks. Taint-tracking has been partially or fully used in other similar approaches [NSS09, Sek09, NtGG$^+$05, NLC07]. Although xJS and RaJa do not rely at all on tainting, a source-code based tainting technique [XBS06] can certainly assist in separating all server-produced JavaScript, mark all legitimate client-side code and also identify malicious data. However, the performance might degrade.

9.1.2 Sophisticated Web Attacks

Through the years, as the web technologies evolved, exploitation of web applications also became more sophisticated. Apart from XSS and CSRF, there are many ways to exploit a web application. By sending specifically crafted data, someone can exploit bugs in an AJAX based client [SHPS] or by including web pages as a Cascade Style Sheet (CSS) someone can steal a user's secret, for example the subjects of her inbox [LSZCC10]. Moreover, even devices such as routers that use web interfaces for configuration can be exploited by injecting specific client-side code [BBB09]. Many of these attacks are recently discovered. There is no explicit categorization of the attacks listed in XSSed.com, which is relevant to these attack flavors. However, we believe that in the future the repository will host reports for web application exploits that use the aforementioned techniques. All, these flavors of web exploitation can in principle use JavaScript transmitted via URLs and, thus, IDSs can take advantage of the JavaScript syntax-tree based signatures presented in Chapter 6.

9.2 Instruction Set Randomization

xJS is based on Isolation Operators and it is inspired by Instruction Set Randomization (ISR) [KKP03]. Solutions based on ISR have been applied to native code and to SQL injections [BK04]. Keromytis discusses ISR as a generic methodology for countering code injections in [Ker09] and he mentions that the technique can be potentially applied to XSS mitigation. However, to the best of our knowledge, there has been no systematic effort towards this approach before. xJS and RaJa, presented in Chapters 4 and 5, are the first systematic efforts in applying ISR to web applications and JavaScript.

9.3 Anomaly Detection Systems and Regular Expressions

Network monitors have been also developed for injections in native code. For example nemu [PAM07] attempts to execute, using emulation, all network traffic in order to identify a shellcode. In a similar fashion, xHunter attempts to parse all parts of URLs, in order to identify XSS exploits. Anomaly detection systems, such as Spectrogram [SKS09] try to separate the benign traffic, which is received by a web application, from the malicious one. In a similar fashion tools such as NoScript [Mao06], NoXSS [Rei08] and Snort [R^+98] use a series of heuristics, based on regular expressions and static signatures to identify malicious web requests. The study presented in Chapter 7 can be beneficial for all these frameworks in two ways. First, the examples of JavaScript exploits listed in Table 7.4 can assist in enhancing the signature set used by these tools. Although, these example exploits serve more as a PoC, they are still valuable, since capturing *attack attempts* is also important. Second, the signatures introduced in this thesis, which are based on JavaScript syntax trees, can assist in building more sophisticated IDSs. Finally, there are arguments supporting that the usage of regular expressions for preventing web exploitation can be considered, under specific circumstances, harmful [BBJ10b]. In a similar fashion, we advocate that regular expressions are not sufficient for capturing all web exploits, because of their limited expressiveness. JavaScript has a rich grammar, which is hard to be expressed by a set of static signatures and regular expressions. On the other hand, a handful of JavaScript syntax trees can express a broad family of web exploits.

9.4 Browser Operating Systems

Recently, there is an effort for applying operating systems' principles in the web browser aiming at the creation of a more secure browser architecture. The research community is trying to transform the web browser into a mini-operating system, which offers web application separation. For the latest proposals we refer the reader to [WFHJ07, WGM$^+$09, GTK08, RG09, TMK10]. All these systems promote a browser architecture that processes web applications like traditional operating systems process native applications. Although, a browser operating system can guarantee that two web applications cannot interfere with each other, it cannot guarantee that a web application is not exploitable. The work outlined in this dissertation can be beneficial for the evolution of these systems. For example, consider the deployment of a service that passively monitors the execution of web applications for possible exploitation, just like xHunter , which is presented in Chapter 6. This service can take advantage of the JavaScript syntax-tree based signatures presented in this Chapter 7. In addition, web applications running in such operating systems can take advantage of ISR-based services, such as the ones presented in Chapters 4 and 5.

9.5 Network Flow Contracts

There is plenty of research in regards to privacy related to the network activities performed by end users. There is a huge effort for designing and implementing complete systems for providing anonymous communications [RR98, FM02, ZZZR05] and some of them have lead to concrete systems that exist in the real world [DMS04, CSWH01]. More recently, researchers identified that privacy leakage can be performed in modern web applications even if the communication channel between the user and the application is encrypted [CWWZ10]. In another paper, the researchers propose a technique for dealing with such leakage [ZLW+10]. NFCs, presented in Chapter 8, review privacy from a different perspective. Instead of providing the design of a system that hides the network traces of a user, we propose a technology that tracks and cryptographically proves all network activity. NFCs guarantee that a user can never be falsely accused by anyone for a network action, as long as the user has never performed this action.

The idea of providing authenticity and confidentiality of network traffic is not new. There are concrete efforts for standards towards this direction, such as IPSec [DH03]. Although the computational overhead of encrypting all traffic is currently restrictive for a massive deployment, researchers have produced fast and practical implementations for encrypting TCP traffic [BHH+10]. However, NFCs do not seek a solution for a wide range of threat models, but rather a practical and efficient solution for a very precise problem. Instead of producing a very elaborate cryptographic protocol, we provide a fast implementation that performs cryptographic signing of every `TCP-SYN` packet, in order to provide guarantees that every network connection is established with the user's verification.

Finally, the idea of querying encrypted data is also not new. Researchers have invented practical techniques for the efficient searching of encrypted content [SWP00]. Dig encryption, presented in Chapter 8, does not aim for just searching encrypted data, but also applying time constraints to the querying process, in order to be resistant against dictionary attacks.

Chapter 10

Conclusion

We have investigated how already proposed approaches for policy enforcement in web browsers, as is, for example, BEEP [JSH07], fail to defeat XSS attacks under specific circumstances. We have presented a new flavor of web attacks, which can defeat all policy frameworks that are based on script whitelisting. This new family of XSS attacks resembles the well-known *return-to-libc* attack and thus we refer to them as *return-to-JavaScript*. During a *return-to-libc* attack, the attacker is forcing a program to execute existing code as it was never designed. For example, she can make an `exec()` call which is already provided by `libc`. In the same fashion, during a *return-to-JavaScript* attack, the attacker is forcing a web application to execute existing whitelisted code as it was never designed. For example, she can execute a function during an `onload` event, which was originally assigned to an `onclick` event. Based on our findings, we proceeded and investigate how ISR, which has been proposed for code-injections in various environments, can be used for mitigation of XSS (including the new *return-to-JavaScript* attacks).

10.1 Using Instruction Set Randomization for XSS Mitigation

In this dissertation we presented two frameworks for XSS mitigation that are based heavily on ISR, xJS and RaJa.

10.1.1 xJS

We presented xJS, a practical framework against the increasing threat of XSS attacks. xJS was motivated by a number of new code-injection attacks that defeat existing approaches, inn particular with the Return-to-JavaScript attacks (see Chapter 3). xJS is a fast and practical way to isolate all legitimate client-side code from possible code injections. The xJS framework suggests the use of Isolation Operators (IO). An IO, such as one that is based on the `XOR` function, aims on *randomizing* all client-side source corpus for protecting it from code injections. All client-side code is transposed to a new domain, in our case the domain defined by the `XOR` function, and thus it is completely isolated from all code injections. Finally, our framework suggests policies expressed as Browser Actions. The web browser executes all trusted client-code after it has been de-isolated bye performing an action, in our case the `XOR` function.

We implemented and evaluated our solution in Firefox and Webkit, and in the Apache web server. Our evaluation shows that (a) every examined real-world XSS attack can be successfully prevented, (b) negligible computational overhead is imposed on the server and browser side, and (c) the user's browsing experience and perceived performance is not affected by our modifications.

10.1.2 RaJa

RaJa is a complete framework for applying ISR directly to JavaScript. RaJa enforces source code randomization by taking care of code interference between JavaScript and other server-side programming languages. All source randomization is carried out using the Mozilla JavaScript engine, SpiderMonkey, augmented with two simple rules to account with code mixing between JavaScript and PHP. To test the effectiveness of the engine in randomizing complex sources, we deploy it in four popular large web applications. RaJa manages to randomize 90.5% of identified JavaScript in approximately half a million lines of code, mixed up with JavaScript, PHP and markup.

We evaluated RaJa in terms of performance and attack coverage. Our analysis suggests that (a) RaJa imposes a fixed overhead of a few tens of milliseconds per page, which is not the dominating overhead (less than a roundtrip in today's Internet), (b) imposes negligible overhead in the browser side and (c) can detect all attacks hosted by XSSed.com, which refer to web sites that are still vulnerable.

10.2 Network Level XSS Detection

In addition, we presented xHunter, a network-level XSS detector that is based on signatures produced by JavaScript syntax trees.

10.2.1 xHunter

xHunter is a tool that is designed for processing large web traces and isolating suspicious URLs that contain XSS attack exploits. xHunter tries to identify parts contained in a URL that produce a valid JavaScript parse tree. If a fragment produces a syntax tree of a certain depth, then the URL is considered suspicious.

xHunter is not meant for providing defenses against XSS attacks. It rather assists in collecting properties for performing studies related to XSS attacks. For example, xHunter can process network traces collected from various sensors distributed all over the world and give answers to questions like *How often web sites are targeted with XSS attacks?* or *Which web sites are the targets?*.

We analyzed all technical challenges concerning the implementation of xHunter and evaluated the prototype. We processed about 11,000 URLs that contain XSS exploits, collected from XSSed.com, and 1,000 benign URLs collected from a monitoring point in an educational organization with about 1,000 users. The results suggest that xHunter has less than 3.2% of false negatives and about 2% of false positives.

10.2.2 JavaScript-based Signatures

Using xHunter we presented an extensive study of 12,000 web attacks hosted in XSSed.com. All these incidents have been recorded in the period of the last three years. Our study aimed at drawing a finer picture about the properties of real web exploits. We showed that even the more frequently accessed web sites, the ones with an Alexa rank below 1,000, are vulnerable to web exploitation. We showed that many web applications that communicate over an encrypted channel can be penetrated using a web exploit. We showed that web vulnerabilities are not always repaired quickly. Finally, we showed that a significant fraction of web exploits is not based on injecting markup elements and thus it is hard to be captured using regular expressions. We performed an analysis based on JavaScript syntax trees in this set of exploits and we created a rule-set with 18 signatures, expressed as a series of JavaScript tokens. This work can be beneficial to network-based IDSs which use more sophisticated engines for identifying malicious patterns.

10.3 Network Flow Contracts

Finally, we addressed problems stemming from the massive collection of digital traces related to every user's daily interaction with the Internet. We formed threat models where fake information injected in a log file can be used as evidence for incriminating an unsuspecting user. We further proceeded to present and evaluate Network Flow Contracts, an architecture that can prove if a network access has actually taken place. An ISP providing NFCs, can guarantee that the network activity of each subscriber has been logged with the user's verification. More precisely, the ISP terminates all connections that have not been digital signed by the subscribers. Furthermore, we presented a minimal version of NFCs, specifically designed for web traffic, which requires the addition of an HTTP header, `X-Signature`, for holding the digital signature of every web request. Finally, we discussed Dig Encryption, a scheme for storing encrypted information that can be resistant to dictionary attacks.

Appendix A

Publications

This thesis has produced several publications in peer-reviewed conferences and journals.

The research associated with exploiting issues in social networks is published in the following publications:

> Elias Athanasopoulos, Andreas Makridakis, Spyros Antonatos, Demetres Antoniades, Sotiris Ioannidis, Kostas G. Anagnostakis, and Evangelos P. Markatos. **Antisocial Networks: Turning a Social Network into a Botnet.** *In Proceedings of the 11th Information Security Conference (ISC 2008).* September 2008, Taipei, Taiwan.

> Andreas Makridakis, Elias Athanasopoulos, Spiros Antonatos, Demetres Antoniades, Sotiris Ioannidis, and Evangelos P. Markatos. **Understanding the Behavior of Malicious Applications in Social Networks.** *In IEEE Network Special Issue on Online Social Networks (September-October 2010).*

The research associated with the *return-to-JavaScript* attacks is published in:

> Elias Athanasopoulos, Vasilis Pappas, and Evangelos P. Markatos. **Code-Injection Attacks in Browsers Supporting Policies.** *In Proceedings of the 3rd Workshop on Web 2.0 Security & Privacy (W2SP).* May 2009, Oakland, California.

The research associated with applying ISR for countering code injections in web applications is published in:

Elias Athanasopoulos, Vasilis Pappas, Antonis Krithinakis, Spyros Ligouras, Evangelos P. Markatos, and Thomas Karagiannis. **xJS: Practical XSS Prevention for Web Application Development.** *In Proceedings of the 1st USENIX Conference on Web Application Development (WebApps).* June 2010, Boston, Massachusetts.

Antonis Krithinakis, Elias Athanasopoulos, and Evangelos P. Markatos. **Isolating JavaScript in Dynamic Code Environments.** *In Proceedings of the 1st Workshop on Analysis and Programming Languages for Web Applications and Cloud Applications (APLWACA), co-located with PLDI.* June 2010, Toronto, Canada.

Elias Athanasopoulos, Antonis Krithinakis, and Evangelos P. Markatos. **An Architecture For Enforcing JavaScript Randomization in Web2.0 Applications (short paper).** *In Proceedings of the 13th Information Security Conference (ISC).* Boca Raton, Florida, October 2010.

The research associated with xHunter is published in:

Elias Athanasopoulos, Antonis Krithinakis, and Evangelos P. Markatos. **Hunting Cross-Site Scripting Attacks in the Network.** *In Proceedings of the 4th Workshop on Web 2.0 Security & Privacy (W2SP).* May 2010, Oakland, California.

Bibliography

[ab] ab - Apache HTTP server benchmarking tool. `http://httpd.apache.org/docs/2.0/programs/ab.html`.

[AKM10] Elias Athanasopoulos, Antonis Krithinakis, and Evangelos P. Markatos. Hunting Cross-Site Scripting Attacks in the Network. In *Proceedings of the 4th Workshop on Web 2.0 Security & Privacy (W2SP)*, Oakland, CA, May 2010.

[AMA$^+$08] Elias Athanasopoulos, Andreas Makridakis, Spiros Antonatos, Demetres Antoniades, Sotiris Ioannidis, Kostas. G. Anagnostakis, and Evangelos P. Markatos. Antisocial Networks: Turning a Social Network into a Botnet. In *Proceedings of the 11th international conference on Information Security*, ISC '08, pages 146–160, Berlin, Heidelberg, 2008. Springer-Verlag.

[Anl02] C. Anley. Advanced SQL injection in SQL server applications. *White paper, Next Generation Security Software Ltd*, 2002.

[APK$^+$10] Elias Athanasopoulos, Vasilis Pappas, Antonis Krithinakis, Spyros Ligouras, and Evangelos P. Markatos. xJS: Practical XSS Prevention for Web Application Development. In *Proceedings of the 1st USENIX WebApps Conference*, Boston, US, June 2010.

[APM09] Elias Athanasopoulos, Vasilis Pappas, and Evangelos P. Markatos. Code-Injection Attacks in Browsers Supporting Policies. In *Proceedings of the 2nd Workshop on Web 2.0 Security & Privacy (W2SP)*, Oakland, CA, May 2009.

[BBB09] Hristo Bojinov, Elie Bursztein, and Dan Boneh. XCS: Cross Channel Scripting and Its Impact on Web Applications. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, pages 420–431, New York, NY, USA, 2009. ACM.

[BBJ10a] Daniel Bates, Adam Barth, and Collin Jackson. Regular Expressions Considered Harmful in Client-Side XSS Filters. In *WWW '10: Proceedings of the 19th International Conference on World Wide Web*, pages 91–100, New York, NY, USA, 2010. ACM.

[BBJ10b] Daniel Bates, Adam Barth, and Collin Jackson. Regular Expressions Considered Harmful in Client-Side XSS Filters. In *Proceedings of the 19th international conference on World Wide Web (WWW)*. ACM New York, NY, USA, 2010.

[BCS09] Adam Barth, Juan Caballero, and Dawn Song. Secure Content Sniffing for Web Browsers or How to Stop Papers from Reviewing Themselves. In *Proceedings of the 30th IEEE Symposium on Security & Privacy*, Oakland, CA, May 2009.

[BHH$^+$10] A. Bittau, M. Hamburg, M. Handley, D. Mazieres, and D. Boneh. The case for ubiquitous transport-level encryption. In *Proceedings of the 19th Conference on USENIX Security Symposium*, 2010.

[BJM08] Adam Barth, Collin Jackson, and John C. Mitchell. Robust Defenses for Cross-Site Request Forgery. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, 2008.

[BK04] Stephen W. Boyd and Angelos D. Keromytis. SQLrand: Preventing SQL Injection Attacks. In *Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference*, pages 292–302, 2004.

[BLMM94] T. Berners-Lee, L. Masinter, and M. McCahill. RFC 1738: Uniform Resource Locators (URL), 1994. `http://www.ietf.org/rfc/rfc1738.txt`.

[chr] Google chromium os. http://www.chromium.org/chromium-os.

[cia] Did the CIA Hijack a Wikileaks Mirror? http://www.infowars.com/did-the-cia-hijack-a-wikileaks-mirror/. Last visited in February 2011.

[Cla] Richard Clayton. Mobile internet access data retention (not!). http://www.lightbluetouchpaper.org/2010/01/14/ mobile-internet-access-data-retention-not/. Last visited in February 2011.

[cr4] Cr-48 Chrome Notebook. http://www.google.com/chromeos/pilot-program-cr48.html.

[CSWH01] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *Designing Privacy Enhancing Technologies*, pages 46–66. Springer, 2001.

[CWWZ10] Shuo Chen, Rui Wang, XiaoFeng Wang, and Kehuan Zhang. Side-Channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 191–206, Washington, DC, USA, 2010. IEEE Computer Society.

[CYK10] Nicolas Christin, Sally S. Yanagihara, and Keisuke Kamataki. Dissecting one click frauds. In *CCS '10: Proceedings of the 17th ACM conference on Computer and communications security*, pages 15–26, New York, NY, USA, 2010. ACM.

[Dam64] F.J. Damerau. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3):171–176, 1964.

[dat] Directive 2006/24/EC on Data Retention. *Official Journal L105, 13/04/2006 P.0054-0063.* http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CELEX:32006L0024:EN:HTML. Last visited in February 2011.

[Des97] S. Designer. Return-to-libc attack. *Bugtraq, Aug*, 1997.

[DH03] N. Doraswamy and D. Harkins. *IPSec: the new security standard for the Internet, intranets, and virtual private networks.* Prentice Hall, 2003.

[DMPW09] S. DiBenedetto, D. Massey, C. Papadopoulos, and P.J. Walsh. Analyzing the Aftermath of the McColo Shutdown. In *Applications and the Internet, 2009. SAINT '09. Ninth Annual International Symposium on*, pages 157 –160, July 2009.

[DMS04] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th Conference on USENIX Security Symposium*, page 21. USENIX Association, 2004.

[DTH06] R. Dhamija, JD Tygar, and M. Hearst. Why Phishing Works. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 581–590. ACM New York, NY, USA, 2006.

[ECM04] E. ECMA. 357: ECMAScript for XML (E4X) Specification. *ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland*, 2004.

[eu-] Written declaration pursuant to Rule 123 of the Rules of Procedure on setting up a European early warning system (EWS) for pedophiles and sex offenders. http://smile29.eu/doc/DS29_EN.pdf. Last visited in February 2011.

[FHEW08] Adrienne Felt, Pieter Hooimeijer, David Evans, and Westley Weimer. Talking to Strangers Without Taking their Candy: Isolating Proxied Content. In *SocialNets '08: Proceedings of the 1st Workshop on Social Network Systems*, pages 25–30, New York, NY, USA, 2008. ACM.

[FM02] M.J. Freedman and R. Morris. Tarzan: A Peer-to-Peer Anonymizing Network Layer. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 193–206. ACM, 2002.

[FP] K. Fernandez and D. Pagkalos. XSSed.com. XSS (Cross-Site Scripting) information and vulnerable websites archive. `http://www.xssed.com`.

[G$^+$05] J.J. Garrett et al. Ajax: A New Approach to Web Applications. *Adaptive path*, 18, 2005.

[GC09] Matthew Van Gundy and Hao Chen. Noncespaces: Using Randomization to Enforce Information Flow Tracking and Thwart Cross-Site Scripting Attacks. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 8-11, 2009.

[GTK08] C. Grier, S. Tang, and S.T. King. Secure Web Browsing with the OP Web Browser. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 402–416. IEEE, 2008.

[Han] R. Hansen. XSS (Cross-Site Scripting) Cheat Sheet. Esp: for filter evasion. `http://ha.ckers.org/xss.html`.

[HG08] R. Hansen and J. Grossman. Clickjacking, 2008.

[irc] IRCache. http://www.ircache.net/.

[JKK06] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *Proceedings of the 27th IEEE Symposium on Security and Privacy*, pages 258–263, Washington, DC, USA, 2006. IEEE Computer Society.

[JLM94] V. Jacobson, C. Leres, and S. McCanne. libpcap, Lawrence Berkeley Laboratory, Berkeley, CA. *Initial public release June*, 1994.

[Jos06] S. Josefsson. RFC 4648: The Base16, Base32, and Base64 Data Encodings, 2006. `http://tools.ietf.org/html/rfc4648`.

[JSH07] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 601–610, New York, NY, USA, 2007. ACM.

[Ker09] Angelos D. Keromytis. Randomized Instruction Sets and Runtime Environments Past Research and Future Directions. In *IEEE Security and Privacy*, number 1, pages 18–25, Piscataway, NJ, USA, 2009. IEEE Educational Activities Department.

[KKP03] G.S. Kc, A.D. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks with Instruction-Set Randomization. In *Proceedings of the 10th ACM conference on Computer and Communications Security*, pages 272–280. ACM New York, NY, USA, 2003.

[Kle] A. Klein. DOM Based Cross Site Scripting or XSS of the Third Kind. Web Application Security Consortium, Articles, 4.7. 2005.

[LAAA06] V. T. Lam, S. Antonatos, P. Akritidis, and K. G. Anagnostakis. Puppetnets: misusing web browsers as a distributed attack infrastructure. In *Proceedings of the 13th ACM conference on Computer and Communications Security*, CCS '06, pages 221–234, New York, NY, USA, 2006. ACM.

[LC06] Lap Chung Lam and Tzi-cker Chiueh. A General Dynamic Information Flow Tracking Framework for Security Applications. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 463–472, Washington, DC, USA, 2006. IEEE Computer Society.

[ldp] LD_PRELOAD Feature. See man page of LD.SO(8).

[LHLHW$^+$04] A. Le Hors, P. Le Hegaret, L. Wood, G. Nicol, J. Robie, M. Champion, and S. Byrne. Document Object Model (DOM) Level 3 Core Specification. *World Wide Web Consortium, Recommendation REC-DOM-Level-3-Core-20040407*, 2004.

[LSZCC10] Huang Lin-Shung, Weinberg Zack, Evans Chris, and Jackson Collin. Protecting Browsers from Cross-Origin CSS Attacks. In *CCS 10: Proceedings of the 17th ACM Conference on Computer and Communications Security*, New York, NY, USA, 2010. ACM.

[MAA$^+$10] Andreas Makridakis, Elias Athanasopoulos, Spiros Antonatos, Demetres Antoniades, Sotiris Ioannidis, and Evangelos P. Markatos. Understanding the Behavior of Malicious Applications in Social Networks. *Netwrk. Mag. of Global Internetwkg.*, 24:14–19, September 2010.

[Mao06] G. Maone. Firefox add-ons: Noscript, 2006. `https://addons.mozilla.org/en-US/firefox/addon/722`.

[Max06] LLC MaxMind. GeoIP, 2006. `http://www.maxmind.com`.

[mca] McAfee: Enabling Malware Distribution and Fraud. `http://www.readwriteweb.com/archives/mcafee_enabling_malware_distribution_and_fraud.php`.

[ML08] Michael Martin and Monica S. Lam. Automatic Generation of XSS and SQL Injection Attacks with Goal-directed Model Checking. In *Proceedings of the 17th USENIX Security symposium*, pages 31–43, Berkeley, CA, USA, 2008. USENIX Association.

[mod] Modify Headers 0.6.9. https://addons.mozilla.org/ en-US/firefox/addon/modify-headers/.

[MS95] D.R. Musser and A. Saini. *The STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library.* Addison Wesley Longman Publishing Co., Inc. Redwood City, CA, USA, 1995.

[NLC07] S. Nanda, L.C. Lam, and T. Chiueh. Dynamic Multi-Process Information Flow Tracking for Web Application Security. In *Proceedings of the 8th ACM/IFIP/USENIX international conference on Middleware*. ACM New York, NY, USA, 2007.

[NSS09] Y. Nadji, P. Saxena, and D. Song. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 8-11, 2009.

[NtGG$^+$05] Anh Nguyen-tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically Hardening Web Applications Using Precise Tainting. In *Proceedings of the 20th IFIP International Information Security Conference*, pages 372–382, 2005.

[One96] A. One. Smashing the stack for fun and profit. *Phrack magazine*, 49(7), 1996.

[PAA$^+$00] S. Pemberton, M. Altheim, D. Austin, F. Boumphrey, J. Burger, AW Donoho, S. Dooley, K. Hofrichter, P. Hoschka, M. Ishikawa, et al. XHTML 1.0: The extensible hypertext markup language, 2000.

[PAM07] Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos. Emulation-based detection of non-self-contained polymorphic shellcode. In *RAID*, pages 87–106, 2007.

[Pax99] Vern Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31(23-24):2435–2463, 1999.

[PMRM08] N. Provos, P. Mavrommatis, M.A. Rajab, and F. Monrose. All your iFRAMES point to us. In *Proceedings of the 17th conference on Security symposium*, pages 1–15. USENIX Association, 2008.

[R$^+$98] M. Roesch et al. Snort. *The Open Source Network Intrusion System. Web page at `http://www.snort.org`*, 1998.

[Rei08] J. Reith. Firefox add-ons: noXSS, 2008. `https://addons.mozilla.org/en-US/firefox/addon/9136`.

[RG09] C. Reis and S.D. Gribble. Isolating web programs in modern browser architectures. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys)*, pages 219–232. ACM, 2009.

[Ric08] L. Richardson. Beautiful Soup-HTML/XML parser for Python, 2008.

[RR98] M.K. Reiter and A.D. Rubin. Crowds: Anonymity for Web Transactions. *ACM Transactions on Information and System Security (TISSEC)*, 1(1):66–92, 1998.

[RSA83] R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM*, 26:96–99, January 1983.

[RV09] William Robertson and Giovanni Vigna. Static Enforcement of Web Application Integrity Through Strong Typing. In *Proceedings of the 18th USENIX Security Symposium*, Montreal, Quebec, August 2009.

[saf] H.R.1076: Internet Stopping Adults Facilitating the Exploitation of Today's Youth (SAFETY) Act of 2009.
http://www.govtrack.us/congress/bill.xpd?bill= h111-1076.

[SAN] SANS Insitute. The Top Cyber Security Risks. September 2009. `http://www.sans.org/top-cyber-security-risks/`.

[SC05] Nicolas Serrano and Ismael Ciordia. Bugzilla, itracker, and other bug trackers. *IEEE Software.*, 22(2):11–13, 2005.

[Sch07] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C.* A1bazaar, 2007.

[SEA$^+$09] Joshua Sunshine, Serge Egelman, Hazim Almuhimedi, Neha Atri, and Lorrie Faith Cranor. Crying Wolf: an Empirical Study of SSL Warning Effectiveness. In *Proceedings of the 18th USENIX Security Symposium*, pages 399–416, Berkeley, CA, USA, 2009. USENIX Association.

[Sek09] R. Sekar. An Efficient Black-box Technique for Defeating Web Application Attacks. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 8-11, 2009.

[Sha07] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: return-into-libc without Function Calls (on the x86). In *CCS '07: Proceedings of the 14th ACM conference on Computer and Communications Security*, pages 552–561, New York, NY, USA, 2007. ACM.

[SHPS] P. Saxena, S. Hanna, P. Poosankam, and D. Song. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)*.

[SKS09] Y. Song, A.D. Keromytis, and S.J. Stolfo. Spectrogram: A mixture-of-markov-chains model for anomaly detection in web traffic. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*, 2009.

[spi] SpiderMonkey (JavaScript-C) Engine. `http://www.mozilla.org/js/spidermonkey/`.

[sun] SunSpider JavaScript benchmark. `http://www2.webkit.org/perf/sunspider-0.9/sunspider.html`.

[SWP00] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 44–, Washington, DC, USA, 2000. IEEE Computer Society.

[Sym] Symantec Corp. April 2008. 1-3. Retrieved on 2008-05-11. Symantec Internet Security Threat Report: Trends for July-December 2007 (Executive Summary).

[t1] RSA Performance of Sun Fire T2000. http://blogs.sun.com/chichang1/entry/ rsa_performance_of_sun_fire. Last visited in February 2011.

[Tec] TechCrunch. AOL Proudly Releases Massive Amounts of Private Data. http://techcrunch.com/2006/08/06/aol-proudly-releases-massive-amounts-of-user-search-data/.

[TH06] B. Tate and C. Hibbs. *Ruby on Rails: Up and Running*. O'Reilly Media, Inc., 2006.

[TLV09] Mike Ter Louw and V.N. Venkatakrishnan. Blueprint: Precise Browser-neutral Prevention of Cross-site Scripting Attacks. In *Proceedings of the 30th IEEE Symposium on Security & Privacy*, Oakland, CA, May 2009.

[TMK10] S. Tang, H. Mai, and S.T. King. Trust and Protection in the Illinois Browser Operating System. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation (OSDI)*. USENIX, 2010.

[Vei04] D. Veillard. Libxml2 project web page. *http://xmlsoft. org*, 2004.

[VMC02] J. Viega, M. Messier, and P. Chandra. *Network security with OpenSSL*. O'Reilly Media, 2002.

[VNJ$^+$07] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceeding of the 14th Annual Network and Distributed System Security Symposium (NDSS)*, 2007.

[WFHJ07] Helen J. Wang, Xiaofeng Fan, Jon Howell, and Collin Jackson. Protection and Communication Abstractions for Web Browsers in MashupOS. In Thomas C. Bressoud and M. Frans Kaashoek, editors, *SOSP*, pages 1–16. ACM, 2007.

[WGM$^+$09] Helen J. Wang, Chris Grier, Alexander Moshchuk, Samuel T. King, Piali Choudhury, and Herman Venter. The Multi-Principal OS Construction of the Gazelle Web Browser. In *Proceedings of the 18th USENIX Security Symposium*, Montreal, Canada, August 2009.

[whi] Web Hacking Incident Database (WHID). `http://www.xiom.com/whid`.

[XBS06] Wei Xu, Eep Bhatkar, and R. Sekar. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *Proceedings of the 15th USENIX Security Symposium*, pages 121–136, 2006.

[xssa] XSS exploit in key example. `http://xssed.com/mirror/33541/`.

[xssb] XXSed.com vulnerability 35059. http://www.xssed.com/mirror/35059/.

[XSSc] XSSed.com. Twitter and Orkut XSS worms in the news. http://xssed.com/news/120/Twitter_and_Orkut_XSS_worms_in_the_news/.

[XSSd] XSSed.com. Web exploit using special encoding. http://xssed.com/mirror/64043.

[ZLW$^+$10] K. Zhang, Z. Li, R. Wang, X.F. Wang, and S. Chen. Sidebuster: automated detection and quantification of side-channel leaks in web application development. In *Proceedings of the 17th ACM conference on Computer and Communications Security*, pages 595–606. ACM, 2010.

[zon] Zone-H. Information about defacements. http://zone-h.org/.

[ZZZR05] L. Zhuang, F. Zhou, B.Y. Zhao, and A. Rowstron. Cashmere: Resilient Anonymous Routing. In *Proceedings of the 2nd Symposium on Networked Systems Design & Implementation-Volume 2*, pages 301–314. USENIX Association, 2005.