KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Isolating and Restricting Client-Side JavaScript

Steven Van Acker

Supervisor:
Prof. dr. ir. F. Piessens
dr. ir. L. Desmet

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

January 2015

Isolating and Restricting Client-Side JavaScript

Steven VAN ACKER

Examination committee:
Prof. dr. ir. H. Hens, chair
Prof. dr. ir. F. Piessens, supervisor
dr. ir. L. Desmet, supervisor
Prof. dr. ir. W. Joosen
Prof. dr. ir. B. Preneel
Prof. dr. A. Sabelfeld
(Chalmers University of Technology, Sweden)
Prof. dr. N. Nikiforakis
(Stony Brook University, USA)

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

January 2015

Uitgegeven in eigen beheer, Steven Van Acker, Celestijnenlaan 200A box 2402, B-3001 Heverlee (Belgium)

ISBN 978-94-6018-951-7

D/2015/7515/7

Acknowledgements

Dear reader,

What you are holding in your hands is a bundle of pages called a doctoral dissertation. It is the final result of four years of scientific research, attempting to make the Web a safer place.

I have often read in the "Acknowledgements" section of dissertations of colleagues, that their work would not have been possible without the support of a great many people. Ready to disprove this assertion, I have long assumed that this was a fallacy and that anything can be accomplished on your own. However, I have slowly come to realize that without the support of these great many people, my life would have become dark and miserable.

These last four years have contained so much more that contributed to this dissertation, but cannot be written in pages or even otherwise expressed. Nevertheless, I will attempt just that.

This research would not have been possible without the financial support of the IWT and iMinds. It is partially funded by the Research Fund KU Leuven, and by the EU FP7 projects STREWS, WebSand, and NESSoS.

To my supervisors Frank Piessens and Lieven Desmet, thank you for allowing me to research topics in this research field. Also thank you for the enthusiasm and guidance that helped improve my work.

To the members of my doctoral jury, thank you for the interesting discussion and feedback that helped improve this text.

To all my colleagues, fellow PhD students and professors: thank you for the fruitful discussions leading to new ideas and insights. More than that, thank you for all the fun and laughs we have shared over the years.

To my colleagues from the administrative personnel and the project office, thank

you for taking care of us so well over the years. Whether it was taking care of crazy requests or sharing funny stories, I have always been able to count on your help.

To my colleagues from systems and network administration, thank you for your support. My research has often caused some strange behavior on both the computer systems and the network. When things went haywire, it would have been all too easy to pull the plug on the research, but you did not. Thank you for your understanding.

I was lucky enough to be able to visit Chalmers University of Technology in Sweden for a three month research project. To my colleagues from the North, thank you for the wonderful time I got to spend in your company. The warm welcome and friendships made me feel very much at home.

To the Hacknam Style CTF team members, thank you for showing me that the next generation is still interested in developing technical skills. Through our CTF participation and workshops, I have learned as much from you as you have from me.

The road walked by a PhD student has a lot of merits, but there are ugly moments too. At times, you have no idea what you are doing, why you are doing them and start to doubt yourself and your choices. At times like this, when the world seems at its darkest, I wish everyone to have beacons of light in their life as my family and friends are in mine.

Thanks to my parents, sisters and brothers for always being there for me and taking my mind off of things when I needed it most. The silliness, teasing and having the comfort of home is something I will always cherish.

Finally, a special thanks to *all* my friends, but in particular Liesje, Sandy, Wim and Peter. Whenever I need to have some fun, I can always count on them to cheer me up and have a good time. For that, I am immensely grateful.

Thank you, all of you.

Steven Van Acker,
January 6th, 2015

Abstract

In today's web applications, no one disputes the important role of JavaScript as a client-side programming language. JavaScript can turn the Web into a lively, dynamic and interactive end-user experience. Unfortunately, JavaScript can also be used to steal sensitive information and abuse powerful functionality.

Sloppy input validation can make a web application vulnerable, allowing malicious JavaScript code to leak into a web application's JavaScript execution environment, where it leads to unintended code execution.

An otherwise secure web application may intentionally include JavaScript from a third-party script provider. This script provider may in turn serve untrusted or even malicious JavaScript, leading to the intended execution of untrusted code.

In both the intended and unintended case, untrusted JavaScript ending up in the JavaScript execution environment of a trusted web application, gains access to sensitive resources and powerful functionality. Web application security would be greatly improved if this untrusted JavaScript could be isolated and its access restricted.

In this work, we first investigate ways in which JavaScript code can leak into the browser, leading to unintended JavaScript execution. We find that, due to bad input validation, malicious JavaScript code can be injected into a JavaScript execution environment through both browser plugins and browser extensions.

Next, we review JavaScript sandboxing systems designed to isolate and restrict untrusted JavaScript code and divide them into three categories, discussing their advantages and disadvantages: JavaScript subsets and rewriting systems, JavaScript sandboxing through browser modifications and JavaScript sandboxing systems without browser modifications. We further research the last two categories, developing and evaluating a prototype of each.

Beknopte samenvatting

In hedendaagse web applicaties betwist niemand de belangrijke rol van JavaScript als een cliënt-zijde programmeertaal. JavaScript kan het Web omzetten in een levendige, dynamische en interactieve ervaring voor de eindgebruiker. Helaas kan JavaScript ook worden gebruikt om gevoelige informatie te stelen en krachtige functionaliteit te misbruiken.

Slordige validatie van invoer kan een web applicatie kwetsbaar maken, waardoor kwaadaardige JavaScript code in een JavaScript uitvoeringsomgeving van een web applicatie kan lekken, waar het leidt tot onopzettelijke uitvoering van code.

Een anders veilige web applicatie kan opzettelijk JavaScript van een derde partij invoegen. Deze derde partij kan op zijn beurt onbetrouwbare of zelfs kwaadaardige JavaScript beschikbaar maken, wat leidt tot de opzettelijke uitvoering van onbetrouwbare code.

In zowel het opzettelijke als onopzettelijke geval, verkrijgt onbetrouwbare JavaScript code die terecht komt in de JavaScript uitvoeringsomgeving van een vertrouwde web applicatie, toegang tot gevoelige middelen en krachtige functionaliteit. Web applicatie beveiliging zou sterk worden verbeterd als deze onbetrouwbare JavaScript kon worden geïsoleerd en diens toegang beperkt.

In dit werk onderzoeken we eerst manieren waarop JavaScript code in de browser kan lekken, hetgeen leidt tot onopzettelijke uitvoering van JavaScript. We ondervinden dat, als gevolg van slechte validatie van invoer, kwaadaardige JavaScript code geïnjecteerd kan worden in een JavaScript uitvoeringsomgeving door zowel browser plugins als browser extensies.

Vervolgens bespreken we JavaScript sandboxing systemen die ontworpen zijn voor het isoleren en beperken van onbetrouwbare JavaScript code, verdelen ze in drie categorieën en bespreken hun voor- en nadelen: JavaScript subsets en herschrijvingssystemen, JavaScript sandboxing via browser wijzigingen en JavaScript sandboxing systemen zonder browser wijzigingen. We gaan dieper in

op de laatste twee categorieën, waarbij we voor elk een prototype ontwikkelen en evalueren.

Contents

Abstract iii

Contents vii

List of Figures xv

List of Tables xix

Abbreviations xxi

1 Introduction 1

1.1 Goals of this thesis 3

1.2 Contributions 4

1.3 Outline of the text 5

2 Background 7

2.1 The World Wide Web and the Internet 8

2.2 Web browsers and web applications 10

2.2.1 The browser security architecture 10

2.2.2 Browser plugins 12

2.2.3 JavaScript 14

2.2.4 JavaScript APIs . . . 15
2.2.5 Browser extensions . . . 16
2.2.6 Browser wars and the World Wide Web Consortium . . . 17
2.2.7 Web applications . . . 18
2.2.8 Sessions and cookies . . . 19
2.3 Web security . . . 20
2.3.1 Trust . . . 20
2.3.2 The Same-Origin Policy . . . 21
2.3.3 Attacker model . . . 23
2.3.4 Third-party script inclusion . . . 24
2.3.5 Cross-site scripting . . . 26
2.3.6 JavaScript injected through plugins and extensions . . . 30
2.4 Conclusion . . . 32

3 `FlashOver`: Automated Discovery of Cross-site Scripting Vulnerabilities in Rich Internet Applications 33

3.1 Introduction . . . 37
3.2 Background . . . 40
3.2.1 Cross-site Scripting . . . 40
3.2.2 Adobe Flash . . . 40
3.2.3 Using SWF files . . . 41
3.2.4 Execution context of SWF files . . . 42
3.2.5 XSS in Flash . . . 42
3.3 FlashOver approach . . . 45
3.3.1 Static analysis . . . 45
3.3.2 Attack URL construction . . . 45
3.3.3 Automated interaction . . . 46
3.4 FlashOver Prototype . . . 46

3.4.1 Static analysis . . . 47
3.4.2 Attack URL construction . . . 48
3.4.3 Automated interaction . . . 51
3.5 Evaluation . . . 52
3.5.1 Experimental setup . . . 52
3.5.2 Results . . . 52
3.5.3 Discussion . . . 54
3.6 Ethical Considerations . . . 55
3.7 Related work . . . 55
3.8 Conclusion . . . 57

4 Monkey-in-the-Browser: Malware and Vulnerabilities in Augmented Browsing Script Markets 59

4.1 Introduction . . . 62
4.2 Greasemonkey . . . 64
4.2.1 Greasemonkey engine . . . 64
4.2.2 Greasemonkey scripts . . . 65
4.2.3 Attack surface . . . 68
4.3 Community-driven script markets . . . 69
4.3.1 Userscripts.org . . . 70
4.3.2 Gathered dataset and statistics . . . 71
4.4 Malware assessment . . . 72
4.4.1 Defining and detecting malware in user scripts . . . 73
4.4.2 Userscripts.org issue reporting . . . 74
4.4.3 Malware observations . . . 75
4.5 Targeted attacks . . . 78
4.6 Attacking weak scripts . . . 80

4.6.1 DOM-Based XSS 80
4.6.2 Overly generic `@include` 83
4.6.3 Resulting malicious capabilities 85
4.7 Related work 86
4.8 Conclusion 88

5 JavaScript Sandboxing **91**
5.1 JavaScript subsets and rewriting 93
5.1.1 BrowserShield 99
5.1.2 ADsafe 100
5.1.3 Facebook JavaScript 101
5.1.4 Caja 102
5.1.5 Discussion 103
5.2 JavaScript sandboxing using browser modifications 106
5.2.1 Browser-Enforced Embedded Policies (BEEP) 108
5.2.2 ConScript 110
5.2.3 WebJail 111
5.2.4 Contego 113
5.2.5 AdSentry 114
5.2.6 Discussion 116
5.3 JavaScript sandboxing without browser modifications 118
5.3.1 Self-Protecting JavaScript 120
5.3.2 AdJail 122
5.3.3 Object Views 123
5.3.4 JSand 126
5.3.5 TreeHouse 127
5.3.6 SafeScript 130

5.3.7 Discussion . . . 131
5.4 Conclusion . . . 133

6 WebJail: Least-privilege Integration of Third-Party Components in Web Mashups 135

6.1 Introduction . . . 138
6.2 Background . . . 140
6.2.1 Same-Origin Policy . . . 140
6.2.2 Integration of mashup components . . . 140
6.3 Problem Statement . . . 141
6.3.1 Attacker model . . . 142
6.3.2 Security-sensitive JavaScript operations . . . 142
6.3.3 Least-privilege integration . . . 143
6.4 WebJail Architecture . . . 144
6.4.1 Policy layer . . . 145
6.4.2 Advice construction layer . . . 149
6.4.3 Deep aspect weaving layer . . . 149
6.5 Prototype implementation . . . 150
6.5.1 Policy layer . . . 150
6.5.2 Advice construction layer . . . 150
6.5.3 Deep aspect weaving layer . . . 152
6.6 Evaluation . . . 153
6.6.1 Performance . . . 153
6.6.2 Security . . . 155
6.6.3 Applicability . . . 155
6.7 Discussion and future work . . . 156
6.8 Related Work . . . 157
6.9 Conclusion . . . 161

7 JSand: Complete Client-Side Sandboxing of Third-Party JavaScript without Browser Modifications 163

7.1 Introduction . . . 166
7.2 Problem statement . . . 168
7.2.1 Integrating third-party JavaScript . . . 168
7.2.2 Malicious script inclusion . . . 169
7.2.3 Requirements . . . 170
7.3 JSAND security architecture . . . 170
7.3.1 Architectural overview . . . 170
7.3.2 Under the hood . . . 171
7.4 Prototype implementation . . . 172
7.4.1 Object-capability system . . . 173
7.4.2 Policy-enforcing membranes . . . 174
7.4.3 Security policies . . . 176
7.4.4 Wrapping the DOM . . . 178
7.4.5 Dynamic script loading support . . . 179
7.4.6 Support for legacy scripts . . . 180
7.5 Evaluation . . . 181
7.5.1 Complete mediation . . . 181
7.5.2 Backwards compatibility . . . 181
7.5.3 Performance benchmarks . . . 185
7.6 Related work . . . 187
7.7 Conclusion . . . 189

8 Conclusion 191

8.1 Contributions . . . 192
8.2 Lessons learned: JavaScript sandboxing . . . 193
8.3 Lessons learned: Large-scale experimentation on the Internet . 195

8.4 The future 198
8.5 Concluding thoughts 202

Bibliography **205**

List of publications **231**

List of Figures

2.1 Structure of an URL with scheme, hostname and path, used to retrieve content from a web server. 9

2.2 The eight subsystems of the reference architecture of a web browser, from [GG07]. 10

2.3 YouTube's Flash application embedded in a webpage. 13

2.4 Synthesized model of the emerging HTML5 APIs, from [ARD+11]. 15

2.5 Example of an embedded live Twitter feed (indicated by the rectangle on the bottom right), from [Twi]. 18

2.6 A typical web application with third-party JavaScript inclusion. The web application running in the browser combines HTML and JavaScript from a trusted source, with JavaScript from an untrusted source. 23

2.7 Relative frequency distribution of the percentage of top Alexa websites and the number of unique remote hosts from which they request JavaScript code, from [NIK+12]. 25

2.8 "Greeting" web application, both used as intended (top) and through a reflected XSS attack (bottom). 26

2.9 Flash advertisement on apple.com vulnerable to a XSS, from [AND+12]. 31

3.1 Advertising Banner on apple.com vulnerable to Cross-site Scripting through Flash. 43

3.2 Schematic overview of our FlashOver prototype: During static analysis, the SWF file is decompiled and regular expressions uncover *potentially exploitable variables* (PEVs) from the ActionScript source-code. These PEVs are inserted into injection templates in the attack URL construction step. The attack URLs are loaded in a real browser in the automated interaction step, resulting in a list of discovered XSS vulnerabilities. 47

3.3 The regular expressions, in pseudo-form, used in our FlashOver prototype to match the names of potentially exploitable variables. 48

3.4 Construction process of an attack URL for `http://target.tld/ad.swf` with swfid `ABCDEF`, injection template id `1` and variable `abc` with id `2`. 50

3.5 Results from our FlashOver experiment, shown as a cumulative plot. The amount of SWF files per site found is divided by 10 to match the scale of the other results. 53

4.1 Greasemonkey Script Installation Dialog. 67

4.2 Historical data of `userscripts.org` on user scripts, forum topics and forum posts. 70

4.3 From the 37,893 user scripts for which the @include domain could be analyzed, 29.9% was designed for the Alexa top 3. 71

4.4 Categories of malware found in the 592 scripts labelled as "harmful" in the `userscripts.org` dataset. Almost 80% is harmless. 74

4.5 Cumulative amount of scripts in our datasets that are similar to older scripts of the full dataset. Notice that the verified harmful dataset contains more scripts with higher similarity to other scripts, than the full dataset. 76

4.6 Sources and sinks used in the lightweight static analysis performed to look for DOM-based XSS. 83

5.1 A typical web application with third-party JavaScript inclusion. The web application running in the browser combines HTML and JavaScript from a trusted source, with JavaScript from an untrusted source. 92

5.2 The scope chain during execution of the example in Listing 5.2. In this depiction, the scope chain grows down so that newly pushed objects are at the bottom. 95

5.3 Architectural overview of a setup where a middlebox is used for code verification and transformation, at the web application site and at the client site. 97

5.4 Executing the "setTimeout()" function will send a message from the JavaScript environment to the component implementing timer functionality, which can be intercepted, modified or rejected by a policy enforcement mechanism in a modified browser. 107

5.5 The WebJail architecture consists of three layers: the policy layer, the advice construction layer and the deep aspect weaving layer, from [ARD+11]. 113

5.6 The AdSentry architecture: advertisements are executed in a shadow JavaScript engine which communicates with the Page Agent via the policy enforcer, from [DTLJ11]. 115

5.7 Relationship between the real JavaScript environment and a sandbox. The sandbox can only interact with a Virtual DOM, which forwards it via the policy enforcer to the real DOM. . . . 119

5.8 Overview of AdJail, showing the real page, the shadow page and the tunnel scripts through which they communicate and on which the policy is enforced, from [LGV10]. 123

5.9 TreeHouse architectural overview. Sandboxes consist of Web-Workers with a virtual DOM. Access to this virtual DOM is mediated by broker according to a policy. If access is allowed, the request is forwarded to the real page's monitor, from [IW12]. 128

6.1 The WebJail architecture consists of three layers: The policy layer, the advice construction layer and the deep aspect weaving layer. 145

6.2 Synthesized model of the emerging HTML5 APIs. 146

6.3 Schematic view of deep aspect weaving. 153

7.1 the JSand architecture. Inside the browser, all access from JSand sandboxes to the JavaScript environment is mediated according to server-supplied policies. 171

7.2 Tree of scripts dynamically loaded by Google Maps. 184

List of Tables

3.1 The 10 injection templates used in our implementation. Each injection template matches a certain example occurrence of a exploitable variable in ActionScript. The injection template indicates what data should be injected for a successful attack. The first template is a control, where the logging URL is injected instead of any code. The other nine inject actual JavaScript code. 49

3.2 Top ten most commonly-named vulnerable variables found in our experiment. 54

4.1 Top five categories to which the "high-profile" user scripts belong, according to the domain for which they were designed. 72

4.2 Five text-strings appearing in ten or more scripts, of which at least 50%, but less than 100% are verified harmful. 77

4.3 Breakdown of amount of scripts with detected DOM-based XSS vulnerabilities according to the used sources and sinks. Only those sources and sinks with any results are shown. (*) the totals do not reflect the sum on each row, but rather the amount of total unique scripts for the given sink. 82

4.4 `@include` and `@match` directive usage, "insecurely" means an overly generic `@include`. 84

5.1 Comparison between prominent JavaScript sandboxing systems using subsets and rewriting systems. 104

5.2 Comparison between prominent JavaScript sandboxing systems using a browser modification. 117

5.3 Comparison between prominent JavaScript sandboxing systems not requiring browser modifications. 132

6.1 Overview of the sensitive JavaScript operations from the HTML 5 APIs, divided in categories. 148

6.2 Function execution overhead. 154

Abbreviations

AJAX	Asynchronous JavaScript and XML
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
AST	Abstract Syntax Tree
BEEP	Browser Enforced Embedded Policies
CERN	European Organization for Nuclear Research (Centre Européen de Recherche Nucléaire)
CORS	Cross-Origin Resource Sharing
CPU	Central Processing Unit
CSP	Content Security Policy
CSRF	Cross-Site Request Forgery
CSS	Cascading Style Sheets
CWE	Common Weakness Enumeration
DNS	Domain Name System
DOM	Document Object Model
ECMA	European Computer Manufacturers Association
FBJS	Facebook JavaScript
FBML	Facebook Markup Language
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
ID	Identifier
IETF	Internet Engineering Task Force

IMDB	Internet Movie Database
IP	Internet Protocol
ISA	Internet Security and Acceleration
IT	Information Technology
JEE	Java Enterprise Edition
JIT	Just In Time
JS	JavaScript
JSON	JavaScript Object Notation
JSONP	JSON with Padding
NCSA	National Center for Supercomputing Applications
OS	Operating System
OWASP	Open Web Application Security Project
PDF	Portable Document Format
RFC	Request for Comments
RFI	Remote File Inclusion
SES	Secure ECMAScript
SME	Secure Multi Execution
SOP	Same-Origin Policy
SPJS	Self-Protecting JavaScript
SQL	Structured Query Language
SSL	Secure Sockets Layer
SWF	ShockWave Flash or Small Web Format
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UI	User Interface
UMP	Uniform Messaging Policy
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VDOM	Virtual DOM
VM	Virtual Machine
VPN	Virtual Private Network

WWW	World Wide Web
XHR	XMLHttpRequest
XML	Extensible Markup Language
XSS	Cross-Site Scripting

Chapter 1

Introduction

Do you remember the early years of the World Wide Web?

Hooking up your computer to a telephone line to dial into an Internet Service Provider, surfing the Internet at 28.8Kbps or equally low speed. A world where every website had a "bookmarks" section, linking to other websites because search engines were not yet up to the task and because maintaining a library of hyperlinks was considered good citizenship on the Internet. A world where not every corporation had an online presence, while those that did were usually plastered with the stereotypical "under construction" banner and accompanying animated GIFs. If this mental image fills you with nostalgia too, then you must also realize how much the Web changed in the past decades.

The young Web is an artifact of the past, and today's Web is very different. It is safe to say that a visitor from the 90s would be baffled by today's Web. Computers, mobile phones, cars, printers, coffee machines, nuclear power plants and even the international space station: the Web is everywhere. Pretty much every organization, no matter how small, has an online presence. In fact, some organizations exist entirely and exclusively on the Web. According to statistics from 2013 Q4, about 39% of the world's population is connected to the Internet [Min]. We use it for realtime news, keeping up to date with our friends, play games, watch online video, shop in online stores and so much more.

None of this would be possible without the technological advancements behind the Web, not in the least due to the development of JavaScript. JavaScript today is no longer that annoying little language on websites, used for useless trinkets. It is now the core of almost every web application and part of the foundations of the web.

The technical achievements that made the Web great can also be used against itself. With the growth of the Internet came commercial interests such as advertising and online sales, attracting companies trying to make money. This money in turn attracted criminals, committing the cyber-equivalent of breaking-and-entering, fraud, money laundering, and abusing the Internet with other offenses for illegal profit. The rise of social media attracted massive amounts of people to share every aspect of their lives online, in great detail. The databases kept by companies such as Facebook, Twitter, LinkedIn, MySpace and others contain a wealth of private information, allowing among other things, online profiling for more targeted advertising. Needless to say, these databases are juicy targets for online criminals seeking to sell this private data to the highest bidder. In recent years, we have also learned that sinister governmental actors use and abuse the Web for tracking, espionage and mass-surveillance.

How is this possible? Complicated software can have bugs and vulnerabilities. The Internet consists of a set of inter-connected computers, both servers and clients, each running a large amount of such complicated software. Software bugs can often be abused by malicious actors to take control of software running on computers, allowing these actors to steal data stored on those computers and even reprogram these computers to e.g. turn them into surveillance equipment.

Web browsers, the most visible piece of software for an end-user, connect with a multitude of servers to retrieve resources in all forms: HTML, CSS, JavaScript, Flash, Java, video, etc. These resources are then parsed and processed by components in the browser and then typically combined into a functional web application. In order to achieve this, the browser needs to execute millions of lines of code, not just code from the browser itself, but also code retrieved from remote sources.

Vulnerabilities can exist in any part of this code, making browsing the Web a potentially dangerous activity. Vulnerabilities in web applications can lead to JavaScript code execution in the browser, allowing an attacker to access a web application's user's sensitive data and resources.

We can differentiate two categories of JavaScript execution: intended and unintended code execution.

Unintended code can leak into the JavaScript execution environment due to vulnerabilities caused by bad programming practices. Insufficient input validation is such a bad programming practice that can lead to cross-site scripting, if it is output as part of a generated HTML document, or used in the dynamic creation of JavaScript code passed to the "eval()" function.

In 2005, MySpace was vulnerable to a cross-site scripting vulnerability, which allowed an attacker to change MySpace users' profiles by simply having those

users visits the attacker's MySpace profile. The "Samy worm" managed to infect more than one million MySpace users through this vulnerability. The execution of the untrusted Samy worm by many MySpace users, is an example of unintended JavaScript code execution.

Intended code execution, on the other hand, is desired by definition. This does not however imply, that all intended code execution is without unexpected security issues. Third-party JavaScript included in a web application, such as e.g. Google Analytics, Google Maps or jQuery, is intended to be executed. Unfortunately, a web developer has no control over the code that gets executed when it comes from e.g. an untrusted or compromised host.

In 2011, a server hosting the popular qTip2 plugin of the jQuery JavaScript library, was compromised. Attackers added malware to the plugin's JavaScript code. Web applications, including this plugin from that compromised third-party script provider, executed the malware instead and put their users' browsers at risk. The execution of the qTip2 plugin was intended by many web application developers including it in their web application, but they did not know of the malware inside. In this case, the JavaScript execution was intended, but the source was untrusted, since it contained malware.

To avoid that this untrusted code wrecks havoc in a web application, it would be wise to not give it full access to all available browser functionality. Restricting JavaScript code's access to available browser functionality is the focus of *JavaScript sandboxing* research.

1.1 Goals of this thesis

The focus of this thesis is on JavaScript executed on the client-side: the unintended execution of JavaScript as well as the intended execution of untrusted JavaScript.

The first goal is to investigate all the ways through which JavaScript can leak into the browser, leading to unintended execution of JavaScript. Unintended JavaScript execution is undesired and the executing JavaScript code should be treated as untrusted. Investigating all avenues through which JavaScript can find its way into a JavaScript engine in the browser, paints a more complete picture in support of the second goal.

The second goal is to limit the impact of untrusted JavaScript code execution in the browser, by isolating it and restricting its available functionality. Untrusted JavaScript code must be isolated from other code, so that its impact can be distinguished from other JavaScript code. Restricting the functionality available

to untrusted JavaScript, diminishes the potential damage it can cause when it executes.

In this thesis, we only consider JavaScript executed in the browser. We do not look at native code running in the browser, any vulnerabilities it may contain or attacks that target it.

We also do not look at the server-side logic of web applications, or the vulnerabilities and attacks that go with it. In particular, we do not consider persistent and reflected Cross-Site Scripting vulnerabilities because those can be remedied on the server-side.

1.2 Contributions

To accomplish the first goal, we investigated ways through which JavaScript can end up in the JavaScript execution environment. The main contributions for this goal are:

- The design, implementation and evaluation of FlashOver, a system to automatically detect XSS vulnerabilities in Flash applications, illustrating the path by which JavaScript can be injected into a web page through a browser plugin. FlashOver was published and presented at the 7th ACM Symposium on Information, Computer and Communications Security (ASIACCS 2012) [AND+12].
- The review of the Greasemonkey browser extension and its script market, leading to the discovery of DOM-XSS vulnerabilities and a novel powerful type of attack dubbed "Privileged, Global XSS," illustrating ways by which JavaScript can be injected into a web page through a browser extension. This work was published and presented at the 9th ACM Symposium on Information, Computer and Communications Security (ASIACCS 2014) [AND+14].

Instead of investigating all possible browser plugins and extensions, we focused on a single well-known case for each category: the Flash browser plugin and the Greasemonkey browser extension. We believe that other browser plugins and extensions provide similar functionality with equally important consequences: attacker-controlled JavaScript can make its way into the JavaScript execution environment and lead to unintended code execution.

To accomplish the second goal, we researched how JavaScript can be isolated and restricted when it has found its way into the JavaScript execution environment. The main contributions for this goal are:

- A thorough survey and analysis of JavaScript sandboxing techniques and literature, divided into three classes: JavaScript subsets and rewriting mechanisms, JavaScript sandboxing techniques using browser modifications and JavaScript sandboxing techniques without the need for browser modifications.
- The design, implementation and evaluation of WebJail, a JavaScript sandboxing system implemented as a browser modification in Firefox. WebJail was published and presented at the 2011 Annual Computer Security Applications Conference (ACSAC 2011) [ARD+11].
- The design, implementation and evaluation of JSand, a JavaScript sandboxing system which does not require any browser modifications. JSand was published and presented at the 2012 Annual Computer Security Applications Conference (ACSAC 2012) [AVAB+12].

Although the JavaScript sandboxing survey is divided into three categories, we only design and implement a prototype for two of those: JavaScript sandboxing with browser modifications, and JavaScript sandboxing without browser modifications. The third category, JavaScript subsets and rewriting systems, was found too impractical in the long term. Nevertheless, the literature survey and evaluation of our prototypes shows that it is possible to effectively isolate and restrict JavaScript.

1.3 Outline of the text

This dissertation consists of eight chapters in total.

Four of these chapters, Chapters 3, 4, 6 and 7, are based on peer-reviewed, accepted and published papers. These chapters each consist of a short preamble with a summary, publication data and an after-the-fact reflection on the work, as well as the full text of the paper without the bibliography.

The other four chapters, Chapters 1, 2, 5 and 8, are new contributions and have been written solely for this dissertation.

The remainder of this dissertation is structured as follows.

Chapter 2 draws the context in which this work should be viewed. It introduces the basics of Web technology, Web browsers, Web applications and Web security.

Chapters 3 and 4 study methods in which third-party JavaScript can be inserted into webpages through browser plugins and extensions respectively. Chapter 3 describes FlashOver, a system to automatically scan rich Internet applications for XSS vulnerabilities, Chapter 4 investigates malware and vulnerabilities in Greasemonkey, an augmented browsing extension, and its script market.

Chapter 5 discusses research in JavaScript sandboxing, dividing it into three categories: JavaScript subsets and rewriting systems, JavaScript sandboxing through browser modifications and JavaScript sandboxing without browser modifications.

Chapters 6 and 7 describe two JavaScript sandboxing techniques that can isolate JavaScript running in a webpage and limit the available functionality to a user-defined subset. Chapter 6 introduces WebJail, a client-side security architecture to enable least-privilege separation of components in a web mashup through a JavaScript engine modification. Chapter 7 describes JSand, a JavaScript sandboxing system that does not require any browser modifications.

Chapter 8 concludes this dissertation by reviewing the contributions, lessons learned and indicating opportunities for future research.

Chapter 2

Background

"I just had to take the hypertext idea and connect it to the TCP and DNS ideas and –ta-da!– the World Wide Web."

— Sir Tim Berners-Lee,
inventor of the World Wide Web

"When I created JavaScript in May 1995 [in about ten days], my influences were awk, C, HyperTalk, and Self, combined with management orders to 'make it look like Java.' "

— Brendan Eich,
inventor of JavaScript

This chapter draws the context for this text, introducing and explaining basic concepts related to the Web and its associated technologies.

Section 2.1 introduces the Web itself and three fundamental concepts: HTML, HTTP and URLs.

Section 2.2 sketches a short history of web browsers, their architecture and different components. It discusses JavaScript, browser plugins and extensions, the standardization of Web technologies, web applications, sessions and cookies.

Section 2.3 looks at topics related to Web security. It defines the notion of trusted and untrusted code and introduces the Same-Origin Policy. It further focuses on the dangers associated with third-party JavaScript inclusion, Cross-

```
<html>
  <head>
    <title>A simple web page</title>
  </head>
  <body>
    <b>This page showcases the structure of an HTML document</b>
    <p>
      This paragraph contains some text and a hyperlink to
      <a href="http://info.cern.ch">the very first website</a>.
    </p>
  </body>
</html>
```

Listing 2.1: Simple HTML document.

Site Scripting in its different forms and ways in which JavaScript can leak through browser plugins and extensions, into the JavaScript environment.

Finally, Section 2.4 concludes this background chapters and provides an overview of the remainder of this text, in terms described in this chapter.

2.1 The World Wide Web and the Internet

The World Wide Web (WWW or the Web) today is so big that it is commonly confused with the Internet. In reality, the Web is one of many applications that is built on top of the Internet. Another example of an Internet application is E-mail, which was considered the most popular application of the Internet before the Web.

In 1989, motivated by the need for a documentation system and a collaborative workspace [Timb], a team at the European Organization for Nuclear Research (CERN) [cera] led by Tim Berners-Lee created HTML and HTTP, the foundations of today's Web. Their first website, at `http://info.cern.ch`, is still available today and shows the early capabilities of the Web as experienced by users with the technology of the day [CERb].

HyperText Markup Language (HTML) [IETa] is a markup language used to structure web pages using HTML tags. The HTML sourcecode for a simple webpage is shown in Listing 2.1. A piece of text can be marked as being a paragraph by placing it with an opening tag and closing tag . Similarly, text can be marked as bold by enclosing it with and tags. The example also shows a hyperlink, enclosed in <a> tags which indicates a

reference to another webpage which a visitor can follow, typically by clicking on it. The contents of a webpage is placed in the body of an HTML document, indicated by the <body> tags. Meta-information about the webpage, such as the title, is placed inside the <head> tags. To mark a document as containing HTML, the entire HTML document is enclosed with the <html> tag.

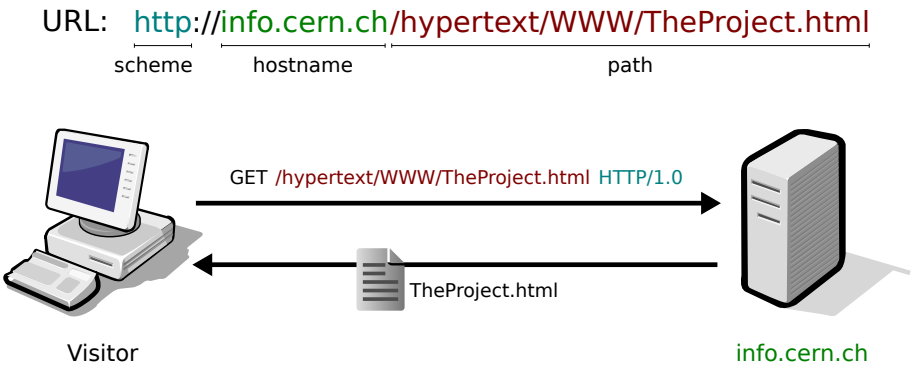


Figure 2.1: Structure of an URL with scheme, hostname and path, used to retrieve content from a web server.

The HyperText Transfer Protocol (HTTP) [IETb] is the communications protocol used between web browsers and web servers to retrieve HTML documents and other content on the Web. Content on the Web can be retrieved from web servers, by pointing a web browser to a Uniform Resource Locator (URL) [IETc] or a "web address." A URL contains all necessary information for a web browser to contact a web server and retrieve the desired content. Figure 2.1 shows the typical structure of a URL and how it is used to retrieve content from the Web. The *hostname* or *domain name* ("info.cern.ch" in the example) indicates *where* a web browser should connect to retrieve the content. The *scheme* ("http" in the example) indicates *how* a web browser should connect to the web server. In particular, which protocol should be used to communicate with the web server. Finally, the *path* ("/hypertext/WWW/TheProject.html" in the example) specifies *what* the web server should return, because it typically serves a lot of web pages and other content.

In addition to these basic URL parts, the following example illustrates other parts including the query string ("?key=value"), the fragment identifier ("#fragmentID"), username and password ("user:pass@") and a portnumber (80, the standard port for an HTTP server): `http://user:pass@example.com:80/index.html?key=value#fragmentID`.

2.2 Web browsers and web applications

The first graphical web browser was released in 1990 by Tim Berners-Lee and was called WorldWideWeb [Tima]. This browser could render web pages as colored text and hyperlinks, but not display images inline. Instead, when an image was present in a web page, the web browser would show an icon. Clicking this icon would cause the image to be downloaded and opened by a helper application.

In 1993, Marc Andreessen and Eric Bina released the NCSA Mosaic [NCS] browser, which was the first web browser able to display images inline in a web page, making web browsing a more pleasant experience.

NCSA Mosaic's technology could be licensed through SpyGlass, allowing the creation of different flavors of web browsers [Eri]. With Mosaic as their common basis and combined with good programming practices, many of today's web browsers share a similar browser architecture.

2.2.1 The browser security architecture

Simply put, a web browser is a computer program used to retrieve content from the Web, interact with it and display it on a screen, either directly or through helper applications. More concretely, a web browser is a complex piece of software comprised of multiple subcomponents, each with its own task, that work together to allow a user to visit the Web.

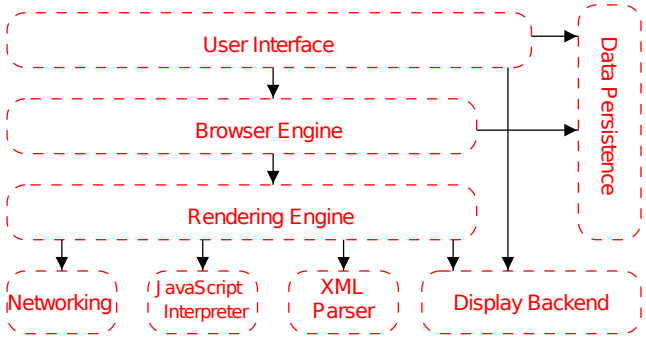


Figure 2.2: The eight subsystems of the reference architecture of a web browser, from [GG07].

The reference architecture of a web browser consists of eight interconnected subsystems [GG07], shown in Figure 2.2:

User Interface.
The part of the browser that interacts directly with the user, displaying windows and toolbars.

Browser Engine.
Handles *Uniform Resource Identifier* (URI a more generic form of URL) loading, and implements browser actions such as the forward and backward button behavior. The browser engine provides a high-level interface to the rendering engine.

Rendering Engine.
The subsystem responsible for displaying content on the screen. It can display HTML and XML, styled with *Cascading Style Sheets* (CSS) and embedding images. It also includes the HTML parser, turning HTML content into the *Document Object Model* (DOM), a structured form more suitable for other components. For the sake of compatibility with older browsers, many HTML parsers also have a *quirks mode* [qui] next to a *standards mode*. In standards mode, the HTML parser strictly complies to W3C and IETF standards and rejects any malformed HTML. In quirks mode however, the HTML parser is more lenient and quietly repairs broken HTML instead of rejecting it.

Networking Subsystem.
The part of the browser responsible for communicating with the network over protocols such as HTTP, loading content from other web servers, caching data and converting data between different character sets.

JavaScript Interpreter.
Also known as the JavaScript engine, this subsystem parses and executes JavaScript code. JavaScript itself is an object-oriented programming language that can evaluate expressions, but does not define ways to influence the rest of the world. To interact with the outside, such as the other browser components, the user or the network, the JavaScript engine must communicate with other subsystems.

XML Parser.
Parses XML documents into a DOM structure. This component is different from the HTML parser and is a generic, reusable component. The HTML parser on the other hand, is optimized for performance and tightly coupled with the rendering engine.

Display Backend.
This component provides an interface to the underlying operating system to draw windowing primitives and fonts.

Data Persistence.
Stores and retrieves data such as browsing history, bookmarks, cookies and browser settings.

The modular subsystems are often reused between different browser vendors. For instance, the Gecko [Moza] browser engine is used by Mozilla Firefox, Netscape Navigator, Galeon [gal] and others. Google Chrome uses the Blink [Bli] browser engine, also used by Opera [Ope] and the Android browser [Webb]. Microsoft Internet Explorer uses the Trident [Micc] layout engine, also used by the Maxthon [Max] browser. Browser components are not only reused by web browsers. Mozilla Firefox's JavaScript engine, SpiderMonkey [spib], is also used in the GNOME3 desktop environment [GNO], and can be used as a standalone JavaScript interpreter. Google Chrome's JavaScript engine, V8 [Gooe], also powers node.js [nod], a server-side JavaScript runtime environment.

Many of these subsystems are used by the browser during routine operations such as loading and rendering a webpage. When a user points a browser to a webpage and the browser has downloaded an HTML document, the *rendering pipeline* is started that will eventually display the webpage and allow the user to interact with it.

The rendering pipeline generally consists of 3 steps: parsing, layouting and rendering:

- During the parsing step, the downloaded HTML document is parsed into a data structure known as the *Document Object Model* (DOM) tree. Each node in this tree comprises an HTML element, with links to the parent element and sub-elements.
- In the layouting step, rectangular representations of the nodes in the DOM are arranged according to the styling rules dictated by the webpages and its *Cascading Style Sheets* (CSS) information.
- Finally, in the rendering step, a graphical representation of each HTML element in the DOM is painted in its respective rectangular representation, and finally drawn onto the user's screen.

This rendering pipeline is a gradual process that is re-iterated while a browser loads all the needed resources.

2.2.2 Browser plugins

Many of NCSA Mosaic's developers went on to work for Netscape [netb], founded by Marc Andreessen and James Clark, and created the Netscape Navigator

browser. Netscape envisioned the web as a distributed operating system [Cha], with code running on both web servers and in web browsers. Netscape Navigator 2.0 supported this vision by allowing Java applets to be displayed inline in a web page through the use of a novel feature: a *browser plugin* [neta].

Browser plugins are code modules, normally compiled to native code, which extend the browser to provide new functionality. In the case of the Java Applet plugin, it provides the browser with the means to execute Java Applets. Unlike helper applications, browser plugins are not standalone programs. They have access to a web browser API that they can use to interact with the web browser and the content it is displaying. The Adobe Flash plugin, originally created by Macromedia in 1996, is a popular example of a browser plugin in use today.

Plugins can display interactive content inline in a web page, interact with the user, communicate with and manipulate data in web pages. The communication between webpages and plugins goes in both directions. For instance, a website like YouTube can instruct the Flash plugin to automatically start playing a media file. Likewise, a Flash application has the ability to contact the hosting page to e.g. retrieve form data.

Figure 2.3: YouTube's Flash application embedded in a webpage.

Figure 2.3 shows an example of a YouTube's Flash application embedded in a webpage. The Flash plugin will be discussed further in Chapter 3.

In addition, plugins can implement custom functionality that can be exposed to the JavaScript environment. Google Hangouts [Gooc], an online video-

```
1 var name = prompt("What is your name?");
2 var year = prompt("What year were you born?");

3 var today = new Date();
4 var age = today.getFullYear() - year;

5 alert("Hello "+name+", you are about "+age+" years young");
```

Listing 2.2: Example JavaScript code prompting the user for name and birthyear, calculating age and displaying it in a pop-up.

conferencing application, requires a plugin in older browser to access a computer's webcam and microphone. This plugin provides a JavaScript API so that the Google Hangouts web application can access the webcam and microphone from the JavaScript environment.

2.2.3 JavaScript

In 1995, Netscape management told Brendan Eich to create a programming language to run in the web browser that "looked like Java." He created JavaScript in only 10 days [Cha]. In addition to browser plugins, JavaScript was another novel feature of Netscape Navigator 2.0 that supported Netscape's vision of the Web as a distributed operating system. In contrast with Java, which was considered a heavyweight object-oriented language and used to create Java applets, JavaScript would be Java's "silly little brother" [neta], aimed towards non-professional programmers who would not need to learn and compile Java applets.

Listing 2.2 shows a simple example of JavaScript. When executed, the code will prompt for the user's name and birth-year. It will then calculate the user's age based on the current year and display it with a greeting using a pop-up. This JavaScript example makes use of the "prompt()" function, the "Date" object and the "alert()" function.

When an HTML document is about to be loaded, and before the rendering pipeline starts, the browser initializes an instance of the JavaScript engine and ties it uniquely to the webpage about to be loaded.

The webpage's developer can use JavaScript to interact with this rendering pipeline by including JavaScript in several ways. JavaScript can be executed while the pages is loading, using HTML <script> tags. These script tags can

cause the browser to load external JavaScript and execute them inside the webpage's JavaScript execution environment. Script tags can also contain inline JavaScript, which will equally be loaded and executed. HTML provides a way to register JavaScript event handlers with HTML elements, which will be called when e.g. an image has loaded, or the user hovers the mousepointer over a hyperlink. In addition, JavaScript can register these event handlers itself by querying and manipulating the DOM tree. Events are not only driven by the user, but can also be driven programmatically. For instance, JavaScript has the ability to use a built-in timer to execute a piece of JavaScript at a certain point in the future. Likewise, the *XMLHttpRequest* functionality available in the JavaScript engine allows a web developer to retrieve Internet resources in the background, and execute a specified piece of JavaScript code when they are loaded. Lastly, JavaScript has the ability to execute dynamically generated code through the *eval* function.

2.2.4 JavaScript APIs

JavaScript's capabilities inside a web page are limited to the APIs that are offered to it. Typical functionality available to JavaScript in a web page includes manipulating the DOM, navigating the browser and accessing resources on remote servers.

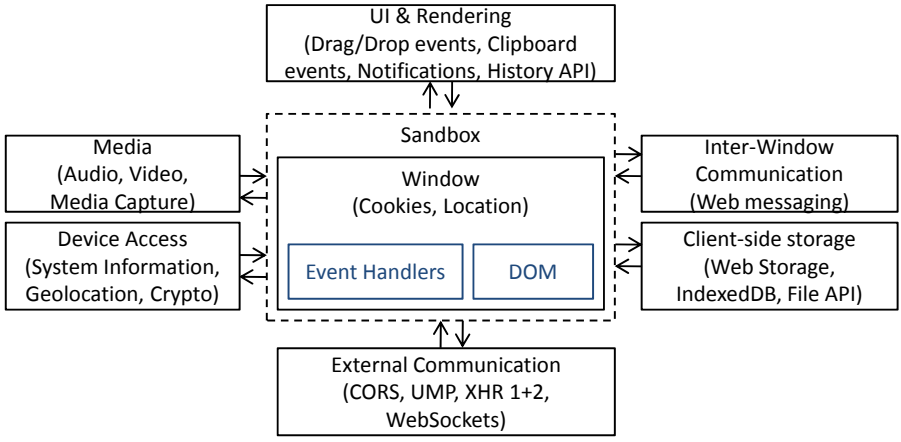


Figure 2.4: Synthesized model of the emerging HTML5 APIs, from [ARD+11].

In the new HTML 5 and ECMAScript 5 specifications, JavaScript gains access to more and powerful APIs. Figure 2.4 [DRDPP11] shows a model of some of these new HTML 5 APIs, which are further explained below and in Chapter 6.

Inter-frame communication.
facilitates communication between windows (e.g. between mashup components). This includes window navigation, as well as Web Messaging (postMessage).

Client-side storage.
enables applications to temporarily or persistently store data. This can be achieved via Web Storage, IndexedDB or the File API.

External communication.
features such as CORS, UMP, XMLHttpRequest level 1 and 2, WebSockets, raw sockets and Web RTC (real-time communication) allow an application to communicate with remote websites.

Device access.
allows the web application to retrieve contextual data (e.g. geolocation) as well as system information such as battery level, CPU information, ambient sensors and high-resolution timers.

Media.
enable a web application to play audio and video fragments, capture audio and video via a microphone or webcam and manage telephone calls through the Web Telephony API.

The UI and rendering.
allow subscription to clipboard and drag-and-drop events, issuing desktop notifications, allow an application to go fullscreen, populating the history via the History API and create new widgets with Web Components API and Shadow DOM.

2.2.5 Browser extensions

Browser extensions are JavaScript applications running inside a privileged environment in a web browser. Unlike JavaScript running inside a web page, JavaScript running in a browser extension has access to more powerful browser APIs allowing privileged operations such as adding menus or accessing other web pages in the same browser. Just like browser plugins, browser extensions can expose functionality to the JavaScript environment, making it available to JavaScript running in a web page. Because they are written in JavaScript, extensions do not require a compilation step to native code and are thus easier to develop than browser plugins.

An example of a browser extension is AdBlock [adb], which is the most popular browser extension for Firefox [Mozc] and Chrome [Gooa] as of this writing. AdBlock detects and removes advertisements from web pages on the client-side,

by monitoring all web pages loaded into the browser. In addition to removing advertisements from web pages, AdBlock also removes advertising from other content, e.g. YouTube video clips. Without the special privileges assigned to browser extensions, AdBlock would not be able to monitor or alter all web pages in the browser.

Another example is Greasemonkey [gre]. Greasemonkey allows users to execute custom JavaScript snippets, named *user scripts*, on any web page they load in their browser. While Greasemonkey itself is a browser extension and runs in a privileged environment in the browser, user scripts running inside of Greasemonkey are subject to more restrictions imposed by Greasemonkey. Still, Greasemonkey scripts can perform interesting tasks, such as helper scripts for online games to automate in-game tasks. Chapter 4 provides more detail on Greasemonkey.

2.2.6 Browser wars and the World Wide Web Consortium

The combination of Tim Berners-Lee's HTML and HTTP, combined with the release of NCSA's Mosaic browser started the boom of the World Wide Web [ZDN]. More than 120 companies licensed Mosaic's technology through SpyGlass, which had licensed Mosaic itself [Eri], some with the intention to build their own browsers.

To compete with other browser vendors in a period known as "the first browser wars," a new browser needed to have unique features that set it apart from the others. Netscape, a new company founded with many of the original developers of Mosaic, introduced the <blink> tag [Lou] into the Netscape Navigator browser. The blink tag caused text enclosed in this tag to alternate between shown and hidden. Microsoft, licensing SpyGlass Mosaic with its Internet Explorer browser, introduced the <marquee> tag [Micd], which caused enclosed text to scroll across the screen. Microsoft never implemented Netscape's blink tag [Mica], although Mozilla Firefox (Netscape Navigator's successor) eventually implemented the marquee tag to be compatible with Internet Explorer [Mozd]. In addition to new HTML tags, Microsoft Internet Explorer also introduced JScript, a dialect of JavaScript compatible with Netscape's JavaScript.

The browser wars caused browser vendors to invent new, non-standard and proprietary extensions to the HTML standard, which confused web developers to the point where they would create a separate website for a particular browser [W3Cf]. In an effort to standardize web technology, Tim Berners-Lee founded the *World Wide Web Consortium* (W3C) in 1994. In 1996, ECMA standardized Netscape's JavaScript and Microsoft's JScript into ECMAScript.

The W3C promotes the development of high-quality standards based on the consensus of the community. W3C processes promote fairness, responsiveness, and progress [W3Ca]. Currently, the W3C has 396 members which include all major browser vendors, as well as experts from industry and academia [W3Cc].

2.2.7 Web applications

A web application combines HTML code, JavaScript and other resources from several web servers, into a functional application that runs in the browser. Unlike typical desktop applications which need to be installed on a computer's hard disk, web applications are accessible through the web browser from anywhere and do not need to be installed.

A key component in today's web application, is JavaScript. JavaScript code in a web application executes in the browser and can communicate with a web server, which typically also executes code for the web application.

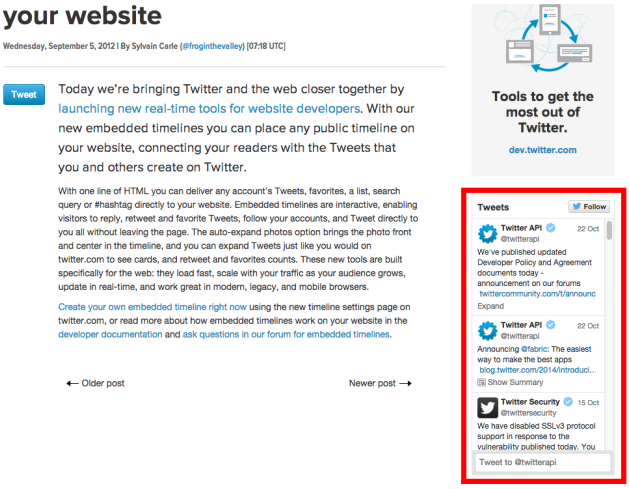


Figure 2.5: Example of an embedded live Twitter feed (indicated by the rectangle on the bottom right), from [Twi].

Consider a website wishing to display the latest tweets from a Twitter feed. Such a widget can be embedded into a web page, as shown in Figure 2.5. Without a client-side programming language such as JavaScript, the web server from which this web page is retrieved, could gather and insert the latest tweets at the moment the web page was requested, and insert them into the web page as HTML-formatted text. When rendered, the visitor would see the latest tweets,

but they would not update themselves in the following minutes because the web page is static.

Another option is to use JavaScript on the client-side. When the web page is requested, the web server can insert JavaScript that regularly requests the latest tweets from the feed and updates the web page to display them. The result is an active web page that always displays the latest information.

This example consists of only one HTML page and requests information from one source. Today's web has many web applications combining a multitude of third-party resources. Examples are Facebook, YouTube, Google Maps and more.

2.2.8 Sessions and cookies

Web applications frequently require that a user authenticate before being allowed access. This authentication step is most often completed by a user logging in with username and password [BHvOS12].

HTTP however, is a stateless protocol: once a request has been made to a web server, and a request has been received, the connection to that web server may be closed. Subsequent requests would be isolated events and would require the user to re-authenticate to the web server.

To avoid this re-authentication overhead, a web application can establish a *session* with the user. During the lifetime of a session, the web server recognizes the user across different HTTP requests, allowing to keep a per-user state e.g. remembering that the user is authenticated. A session can be maintained until either the user or the web application terminates it.

HTTP has a built-in state management mechanism [IETd] called *cookies*. HTTP cookies are pieces of data that a web server can send to a web browser through the "Set-Cookie" HTTP header. The web browser stores this cookie and submits it to the web server on every subsequent request through the "Cookie" HTTP header.

To establish a session with a user, a web application creates a *session cookie*, which is a cookie with a unique identifier. On every request, the user sends the session cookie to the web application, allowing the web application to recognize him.

JavaScript has access to the cookies for the web application in which it executes, allowing to read, write or delete them through the DOM's "document.cookie" property. To prevent JavaScript from tampering with HTTP cookies, a web

server can mark a cookie with the *HttpOnly* attribute [Mich], instructing the browser to make them inaccessible to JavaScript.

It is important to realize that a session cookie should be kept private. An attacker who manages to get a copy of a user's session cookie, can impersonate that user to the web application by taking over the session. Such a scenario is called a *Session Hijacking Attack* [OWAd].

Other attacks on web browser session management, such as Cross-Site Request Forgery (CSRF), are out of scope for this text. We refer to De Ryck's PhD text [DR14] for a detailed discussion on this research area.

2.3 Web security

Both web browsers and web applications consist of complicated software. The combination of web browsers, web applications and resources from multiple sources creates even more complexity that is prone to mistakes. Some of these mistakes can lead to exploitable vulnerabilities that allow attackers to take control of the JavaScript environment in which a web application executes.

2.3.1 Trust

This text frequently uses the terms "trusted" and "untrusted." These terms are meaningless unless they can be defined in the context of Web security.

In the field of psychology, trust is clarified as follows:

> "Trust is a subjective assessment of another's influence in terms of the extent of one's perception about the quality and significance of another's impact over one's outcomes in a given situation, such that one's expectation of, openness to, and inclination toward such influence provide a sense of control over the potential outcomes of the situation." [Rom03]

Unfortunately, this definition is rather vague and not of much practical use in this research field. In fact, using the word "trust" is considered dangerous in the field of computer security, because of its many different and often contradictory meanings [Gol06].

This section does not aim to define a one-size-fits-all definition of "trust," but rather attempts to convey how the different terms related to "trust" should be

interpreted when used in this text. The terms *trusted code* and *untrusted code* in this text are used consistently with the same meaning.

A *trusted component* is one that can "hurt" you, be it in terms of security, safety, privacy, reputation, money or other. This notion of trust seems counter-intuitive: surely the previous sentence should say that an *untrusted component* can hurt us and that a *trusted component* cannot? No.

If any component *could not* hurt us, there would be no reason to trust it, distrust it or consider it at all.

Of all the components that *can* hurt us, we have assessed certain components and believe that they *will not* hurt us. Those components are *trusted components*.

Note that trust is not a binary concept where one either completely trusts or completely distrusts. Between trusted and untrusted are many gradations. Trust and distrust can increase or decrease depending on e.g. experience or evidence. A component is considered *trustworthy* if it comes with evidence that it can be trusted.

When it comes to software, *trusted code* is code that runs with many privileges because we deem it trustworthy and trust that it will not abuse those privileges to hurt us. *Untrusted code* on the other hand, is considered untrustworthy. Since we do not trust it enough to believe that it will not hurt us, we assume the worst: that it will hurt us or be coerced by an attacker in order to hurt us. Untrusted code should therefor be given as little privileges as possible, and isolated from trusted code as much as possible.

The meaning of other terms related to trust, such as "trusted source", "trusted JavaScript" or "trusted web application" should be interpreted in the same spirit.

2.3.2 The Same-Origin Policy

If web applications were allowed complete access to a browser, they would be able to interfere in the operation of other web applications running in the same browser. Given the powerful APIs briefly discussed in the previous section, a web application would be able to access another web application's DOM, local storage and data stored on remote servers.

To prevent this, web applications are executed in their own little universe inside the web browser, without knowledge of each other. The boundaries between these universes are drawn based on the *Same-Origin Policy* (SOP) [W3Ce].

When the root HTML document of web application is loaded from a certain URL, the *origin* of that web application is said to be a combination of the scheme, hostname and port-number of that URL. For instance, a web application loaded from `https://www.example.com` has scheme *https*, hostname *www.example.com* and, in this case implicit, port number *443*. The origin for this web application is thus *(https,www.example.com,443)* or `https://www.example.com:443`.

The *Same-Origin Policy* (SOP) dictates that any code executing inside this origin only has access to resources from that *same* origin, unless explicitly allowed otherwise by e.g. a Cross-Origin Resource Sharing (CORS) [W3Ch] policy. In the previous example, the web application from `https://www.example.com:443` cannot retrieve the address book from a webmail application with different origin `https://mail.example.com:443` running in the same browser, unless the latter explicitly allows it.

The same-origin policy is part of the foundation of web security and is implemented in every modern browser. In this text we only consider the restrictions imposed by the SOP on the execution of JavaScript.

Insecurely written web applications may allow attackers to breach the same-origin policy by executing their JavaScript code in that web application's origin. Once arbitrary JavaScript code can be injected into a web application, it can take over control and access all available resources in that web application's origin.

Consider a typical webmail application, such as Gmail, allowing an authenticated user to access his emails and contact list. The webmail application offers a user interface in the browser and can send requests to the webmail server to send and retrieve emails, and manipulate the contact list.

An attacker may manage to lure an authenticated user of this webmail application onto a specially crafted website. This website could try to contact the webmail server to send and retrieve emails and contact information, just as the web application would. However, the webmail application's origin is e.g. `https://webmail.com:443`, while the attacker's website is `https://attacker.com:443`. Because of the SOP, JavaScript running on the attacker's website has no access to resources of the webmail's origin.

Now consider what would happen if the webmail application is written insecurely, so that an attacker can execute JavaScript in its origin: `https://webmail.com:443`. Because the attacker's code runs inside the same origin as the webmail application, it has access to the same resources and can also read and retrieve emails and contact information. Because of the power of JavaScript, an attacker can do much more. Specially crafted JavaScript can compose spam email messages and send them out using the victim's email account, or it could erase

the contact list. It could even download all emails in the mailbox and upload them to another server.

An attacker with the ability to execute JavaScript in a web application's origin can take full control of that web application. In the typical web application scenario, untrusted JavaScript can be executed in two ways: by including it legitimately from a third party, or by having it injected through a Cross-Site Scripting vulnerability in the web application or an installed browser plugin or extension.

2.3.3 Attacker model

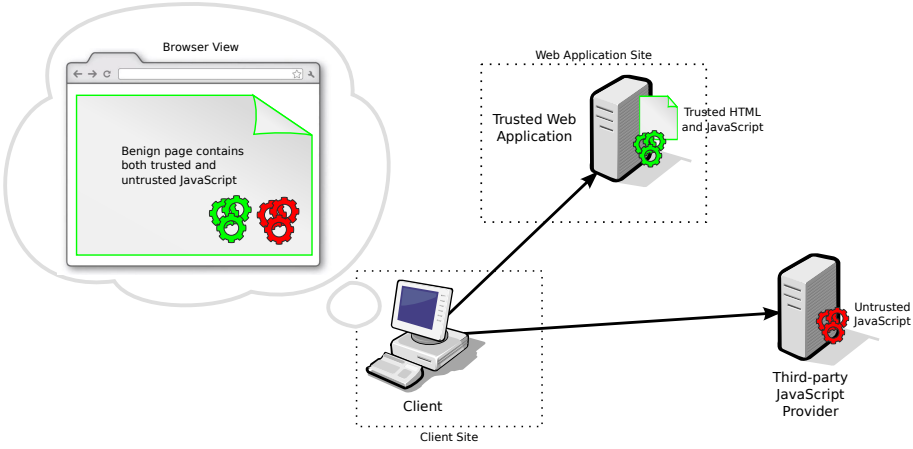


Figure 2.6: A typical web application with third-party JavaScript inclusion. The web application running in the browser combines HTML and JavaScript from a trusted source, with JavaScript from an untrusted source.

When discussing Web security, it is important to keep in mind a typical web application with third-party JavaScript and the actors involved in it. Figure 2.6 shows such a typical web application where HTML and JavaScript from a trusted source are combined with JavaScript from an untrusted source. Remember that all JavaScript, trusted or untrusted, running in a web application's origin has access to all available resources.

There are three actors involved in this scenario: The developer of the trusted web application and the server it is hosted on, the developer of the third-party JavaScript and the server it is hosted on, and the client's browser.

Both the client and the trusted web application have a clear motive to keep untrusted JavaScript from accessing the web application's resources. The client will wish to protect his own account and data. The trusted web application has its reputation to consider and will protect a user's account and data as well. Furthermore, the client does not need to steal information from himself and can use any of his browser's functionality without needing to use a remote web application. Likewise, the web application developer owns the origin in which the web application runs. Stealing data from his own users through JavaScript is not necessary.

It may be the case that the client has modified his browser and installed a browser plugin or extension. Such a plugin or extension may be designed to make the interaction with the web application easier or automated, potentially circumventing certain defensive measures put in place by the developer of the web application. In this scenario, the client is still motivated to protect his account and data, but may be exposing himself to additional threats through the installed browser plugins or extensions that form additional attack surface.

The third-party script provider however, does not necessarily share the same desire to protect a user's data. Even with the best of intentions, a third-party script provider may be compromised and serving malicious JavaScript without its knowledge. It may be the case that the script provider has an intrusion-detection system in place that will detect when it is serving malware, but this would be wishful thinking. In the worst case, the third-party script provider is acting maliciously on its own for whatever sinister reason. In any case, the client and trusted web application cannot trust a third-party script provider with their secrets.

The attacker model best associated with this actor is the *gadget attacker* [BJM09]. A gadget attacker is a malicious actor who owns one or more machines on the Internet, but can neither passively not actively intercept network traffic between the client's browser and the trusted web application. Instead, the gadget attacker has the ability to have the trusted web application's developer integrate a gadget chosen by the attacker.

2.3.4 Third-party script inclusion

Web applications are built from several components that are often included from third-party content providers. JavaScript libraries like jQuery or the Google Maps API are often directly loaded into a web application's JavaScript environment from third-party script providers.

In a large-scale study of the Web in 2012 [NIK+12], we found that 88.45% of

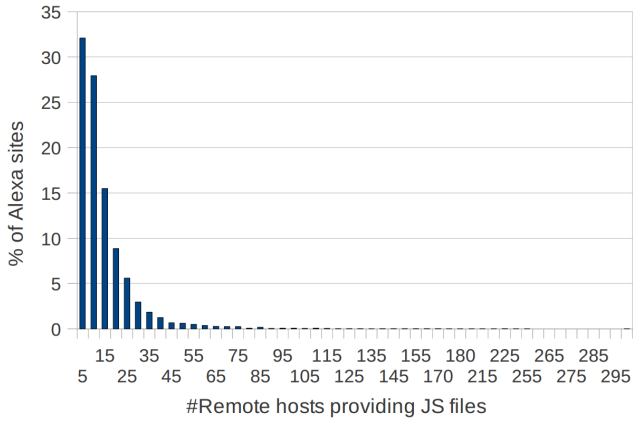


Figure 2.7: Relative frequency distribution of the percentage of top Alexa websites and the number of unique remote hosts from which they request JavaScript code, from [NIK+12].

the top 10,000 web sites on the Web, include JavaScript from a third-party script provider. Figure 2.7 shows the distribution of the number of third-party script providers each web site includes. While about a third include JavaScript from at most 5 remote hosts, there are also web sites that include JavaScript from more than 295 different remote hosts.

Including JavaScript from remote hosts implicitly trusts these hosts not to serve malicious JavaScript. If these third-party script providers are untrustworthy, or if they have been compromised, a web application may end up executing untrusted JavaScript code.

As an example, consider jQuery, a popular multi-purpose JavaScript library used on 60% of the top million websites on the Web [Bui]. The host distributing jQuery was compromised in September 2014 [jQu], giving the attackers the ability to modify the library and possibly infect many websites that include the library directly from `http://jquery.com`. Fortunately, the attackers did not modify the jQuery library itself, but used the compromised server to spread malware instead. Although the JavaScript library itself was not tampered with, the jQuery compromise indicates the inherent security threat that third-party script inclusions can pose.

2.3.5 Cross-site scripting

In a *Cross-Site Scripting attack* (XSS) [OWAa], attacker-controlled JavaScript code is injected in a web application of another origin, allowing the attacker to take control of that web application.

An XSS attack can be caused by unsanitized or badly sanitized input, or unforeseen injection points of JavaScript into an application. Similarly, because JavaScript has the ability to dynamically compose new JavaScript code and execute it, a web application may end up executing untrusted JavaScript if unsanitized input is involved in the creation of this new JavaScript code.

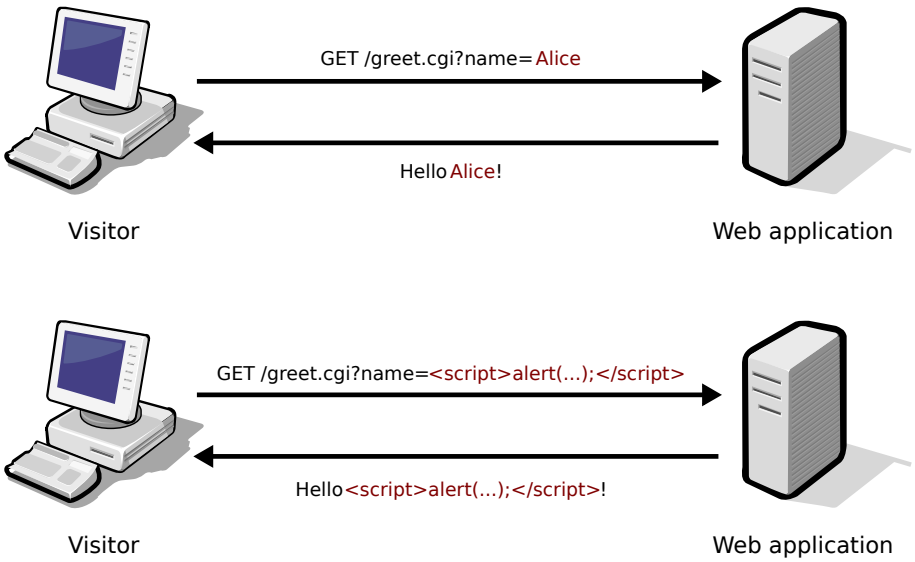


Figure 2.8: "Greeting" web application, both used as intended (top) and through a reflected XSS attack (bottom).

An example XSS attack, more specifically a Type-II or reflected XSS attack, is illustrated in Figure 2.8. In this example, the "Greeting" web application simply prints out a greeting. A user of this web application is prompted for her name through a form, which is then sent to the web server. The web server generates HTML on the server-side, treating the supplied input as a parameter in the generated output.

The expected case is as follows: A request sent by the browser after the user has entered her name, Alice, is shown in Figure 2.8 as the first request. The

```
<html>
  <body>
    Hello Alice!
  </body>
</html>
```

Listing 2.3: Generated HTML output after expected usage of the “Greeting” web application.

```
<html>
  <body>
    Hello <script>alert(document.cookie);</script>!
  </body>
</html>
```

Listing 2.4: Generated HTML output after a reflected XSS attack on the “Greeting” web application.

resulting HTML code generated by the web server is shown in Listing 2.4, showing “Alice” in the output.

Good coding practice is to validate all inputs [OWAb], by e.g. only allowing alphanumeric characters. Consider what happens if the example web application does not correctly validate its input.

Because the input appears directly in the output generated by the web application’s HTML code, an attacker can supply HTML code as input, which will appear in the generated HTML code. The browser receiving this HTML code will parse and render it, expecting all of it to be part of the web application. In reality, the received HTML code contains attacker code, also called the *payload* of the attack, which is executed in the web application’s origin and gaining access to its resources.

This scenario is illustrated by the second request in Figure 2.8, resulting in the HTML code listed in Listing 2.4. In this case, the attacker supplies a piece of code to pop up a window showing the user’s session cookie. In a real-world scenario, an attacker could steal that cookie by sending it to another web server and use it to compromise the victim’s web application account.

In October 2005, the social networking site MySpace was affected by a XSS

vulnerability. Kamkar created a XSS worm [Com] which would add the text "... and Samy is my hero." to a victim's online profile. In addition, the worm submitted a friend request towards Samy's account, resulting in about a million affected MySpace users to inadvertently befriend him.

According to OWASP, XSS is the most prevalent web application security flaw, ranking it third in its 2013 Top 10 [owac]. XSS is ranked fourth in CWE/SANS top 25 most dangerous software errors of 2011 [SAN].

There are three well-known types of XSS attacks: persistent XSS, reflected XSS and DOM-based XSS.

Persistent or type-I XSS

A persistent XSS vulnerability in a web application allows an attacker to store a piece of JavaScript inside the web application. When a user visits the web application, the injected JavaScript code is served as part of the web application's code, thus executing in its origin.

Compared to the previously discussed XSS attack in the "Greeting" web application, a persistent XSS attack would not require the payload to be sent to the web server each time. Instead, the web server keeps it in persistent storage.

Reflected or type-II XSS

A reflected XSS vulnerability allows an attacker to craft a special URL containing JavaScript code encoded within it. When a victim user is tricked into visiting this URL, the server-side logic of the web application processes part of the URL and reflects it back towards the user. The injected code is again served as part of the web application and executed in its origin. The difference with persistent XSS is that in a reflected XSS, the injected JavaScript does not need to be stored on the server.

DOM-Based or type-0 XSS

A DOM-based XSS (or DOM-XSS) allows an attacker to encode JavaScript and store it into the DOM of a webpage that is visited by a victim user. Unlike in a reflected XSS, the injected JavaScript is not processed by the server-side logic of the web application, but by its client-side logic. JavaScript code running in the browser as part of the web application retrieves the injected JavaScript and

outputs a part of it in some form into the DOM. Again, the injected code ends up into the web application's execution context and can access all available resources in its origin.

Lekies et al. [LSJ13] performed a large-scale evaluation of DOM-based XSS vulnerabilities present on the Web, using a taint-aware JavaScript engine. They found that of the top 5000 most popular Internet domains, 9.6% had a web application that was vulnerable to at least one DOM-XSS vulnerability.

XSS countermeasures

```
1 Content-Security-Policy: default-src 'self' *.example.com
```

Listing 2.5: Example CSP header. This policy instructs the browser to only accept JavaScript from its own origin and from subdomains of example.com. By default, inline scripts and "eval()" are not allowed.

The Content Security Policy (CSP) [W3Cb] is a standardized countermeasure for XSS. A web application can use CSP to instruct a browser to only accept JavaScript code from certain sources, not allow the inlining of scripts in HTML, not allow the dynamic execution of JavaScript through "eval()" and more. An example of a CSP policy is shown in Listing 2.5.

Unfortunately, the adoption rate of CSP is very low. Measurements by Weissbacher et al. [WLR14] indicate that only 1% of the top 100 most popular websites uses CSP.

Besides validating user input and CSP, countermeasures such as deDacota [DCJ+13] and ScriptGard [SML11] attempt to resolve XSS by differentiating between valid web application output and user supplied output on the server-side.

Several web browsers, such as Microsoft Internet Explorer [Micb], Google Chrome [Bar], and Mozilla Firefox [Mozh], also implement client-side XSS filters which can detect reflected XSS attacks in the browser. These filters detect and prevent parts of the URL from appearing in the web page.

2.3.6 JavaScript injected through plugins and extensions

Browser plugins and browser extensions, as discussed in Sections 2.2.2 and 2.2.5, can extend the browser and JavaScript environment with extra functionality. While browser plugins are written in native code, browser extensions are written in JavaScript. Both have access to web pages, their DOM and JavaScript environment.

Plugins as well as extensions can have implementation vulnerabilities that can threaten the safe operation of a browser. In this text, we do not consider those types of vulnerabilities. Instead, we regard both plugins and extensions in light of their capability to enhance web applications, since they both have access to a web page's JavaScript environment. We treat plugins and extensions as conduits through which untrusted JavaScript code can leak into a web page's JavaScript environment.

Remember that both plugins and extensions can extend the JavaScript environment with custom functionality that is not necessarily bound by the same-origin policy. JavaScript injected into this extended JavaScript environment would gain access to this functionality, with possibly disastrous consequences.

Consider the example of JavaScript leaking in through the Flash plugin. The Adobe Flash plugin is installed in more than 99% of browsers [fla], allowing advertisers to create interactive animated Flash advertisements to promote services and products. Flash applications are written in ActionScript [Adoc], which is then compiled to create an SWF file that can be opened by the Flash plugin.

```
1 movie 'ad.swf' {
2   button 42 {
3     on (release) {
4       getURL(_root.clickTag, '_blank');
5     }
6   }
7 }
```

Listing 2.6: ActionScript code of a Flash advertisement. Clicking the button will open the URL passed through the "clickTag" parameter.

The ActionScript code of a typical Flash advertisement is shown in Listing 2.6. The code listens for events on a button in the Flash applications. When this button is clicked and released, the "getURL()" function is called with the parameter "_root.clickTag". This parameter can be passed to a Flash

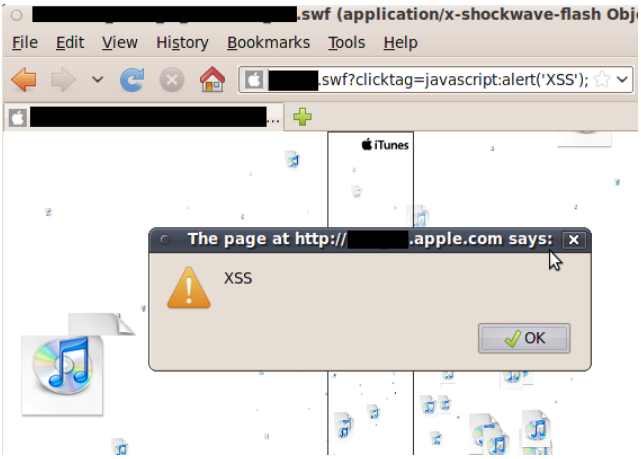


Figure 2.9: Flash advertisement on apple.com vulnerable to a XSS, from [AND+12].

application through the query part of a URL. The "getURL()" method allows a Flash application to navigate the browser to a different page.

Flash applications can also execute JavaScript in a web page's JavaScript environment. In order to inject JavaScript into a web page's JavaScript environment, it is very common [Adoe] to use ActionScript's "getURL()" method with a special JavaScript URI. For instance, the URI `javascript:alert('XSS')` causes the "alert()" function to be executed inside the JavaScript environment, popping up an alert box in the browser.

This special use of ActionScript's "getURL()" method opens up a conduit for an attacker to inject JavaScript into a web page. An attacker can exploit the example advertisement to execute arbitrary JavaScript, by crafting a JavaScript URI and passing it to the "clickTag" query parameter of a Flash application's URL. The result is shown in Figure 2.9 for a vulnerable advertisement in the `http://apple.com` origin.

To determine the prevalence of XSS vulnerabilities in Flash applications, we [AND+12] conducted a large-scale study of the top 1000 most popular websites and found that 64 of them contained such vulnerabilities that were easily exploitable. More details on this issue will be discussed in Chapter 3.

Browser extensions also provide conduits through which JavaScript can be injected into a web page's JavaScript environment. This will be discussed further in Chapter 4.

2.4 Conclusion

This chapter introduced important Web technologies required to understand the remainder of this text.

Web browsers are used to browse the Web and consist of many different cooperating subsystems, such as the HTML parser and the JavaScript engine. Browser functionality can be augmented through browser plugins and browser extensions.

Web applications are a combination of HTML pages, JavaScript code and other resources retrieved from multiple sources running in the browser. Each web application is isolated and protected in its own origin by the same-origin policy.

Badly written web applications, browser plugins and browser extensions may allow attackers to breach the same-origin policy and compromise a web application's JavaScript environment. Discussed security problems include the inclusion of untrusted third-party JavaScript, cross-site scripting and the injection of JavaScript into web applications through browser plugins or extensions.

The following two chapters will explore ways in which JavaScript can leak through browser plugins and browser extensions, into the JavaScript environment of a web page.

Chapter 3 examines the consequences of letting JavaScript leak through a browser plugin like Flash and reports on a large-scale study of the Web to determine how many Flash applications are vulnerable to XSS attacks, by means of FlashOver.

Chapter 4 reveals weaknesses in Greasemonkey and its user scripts that can lead to DOM-based XSS attacks against their users. Since Greasemonkey is a browser extension, this work points out what capabilities an attacker might be able to gain by attacking other browser extensions.

Chapters 5 to 7 will discuss JavaScript sandboxing mechanisms as a way to isolate and restrict untrusted JavaScript code.

Chapter 3

FlashOver: Automated Discovery of Cross-site Scripting Vulnerabilities in Rich Internet Applications

Publication data

Contained in this chapter is the paper titled "FlashOver: Automated Discovery of Cross-site Scripting Vulnerabilities in Rich Internet Applications" as presented at the 7th ACM Symposium on Information, Computer and Communications Security (ASIACCS 2012) [AND+12]. Steven Van Acker was the lead author of this work.

Preamble

This chapter analyzes how attacker-controlled JavaScript can be injected into a web page through a plugin. More specifically, through XSS vulnerabilities present in Flash applications.

This chapter presents FlashOver, a system for automatically analyzing rich Internet applications such as Flash applications and detecting cross-site scripting

vulnerabilities in them.

Flash applications, the most popular rich Internet applications on the Internet [fla, ria], are contained in SWF files which are embedded on web sites and executed through the Flash browser-plugin.

They have access to the JavaScript execution environment of the web page in which they are embedded, and thus run in the same origin as that web page. Flash applications can also be used standalone, without an enclosing web page, in which case an empty web page is generated by the Flash plugin and all code executes in the origin of the web site on which the SWF file is *hosted*.

Flash applications can be passed parameters through HTML attributed when they are embedded in a web page through the "FlashVars" parameter, or via parameters encoded in the URL.

Combining these two features allows an attacker to execute a Flash application in the origin of the hosting web site, with attacker-controlled parameters passed via the URL.

As briefly discussed in Section 2.3.6, a Flash application has several ways to inject JavaScript code in the JavaScript execution context, e.g. through the "getURL()" function. If the Flash application does not sanitize parameters passed through the URL, passing them directly to a sensitive function such as "getURL()", it is possible for an attacker to execute JavaScript code through the Flash plugin on a victim's browser, in the origin of the website on which the SWF file is hosted.

FlashOver is a system that automatically analyzes Flash applications for such XSS vulnerabilities by performing a simple static analysis. FlashOver first decompiles the Flash application to source code using a commercial Flash decompiler. It then statically analyzes the resulting ActionScript code, tracking "sources" such as potential URL parameters into security sensitive "sinks" such as the "getURL()" function. Based on this static analysis, FlashOver constructs attack URLs containing harmless dial-home JavaScript and passes them to an automated interaction module which simulates a user interacting with the Flash application. FlashOver then monitors the server logs where the harmless dial-home JavaScript makes requests to and identifies those Flash applications which are actually vulnerable to the XSS attack.

We tested FlashOver by applying it to all Flash applications we could find on the most popular 1000 Top Alexa domains, resulting in 14,932 SWF files. These SWF files were then decompiled, statically analyzed and automatically interacted with on many computers in parallel, with the help of Mjolnir, an in-house large-scale experimentation framework further described in Section 8.3.

In those 14,932 SWF files, FlashOver found 286 vulnerable Flash applications on 64 Alexa domains, 6 of which belong to the top 50 most popular Internet domains.

The main contributions of this research are:

- Detailed analysis of an XSS attack vector in browser plugins that is commonly overlooked,
- Design and implementation of FlashOver, the first fully automated XSS vulnerability discovery system for Flash applications,
- Evaluation of FlashOver on the top 1000 most popular Internet domains, showing the prevalence of this type of vulnerability in Flash applications and FlashOver's ability to detect it.

In hindsight, FlashOver was the first system that could discover XSS vulnerability in Flash application in an automated way. Despite using a simple static analysis and its limited automated interaction capabilities, FlashOver was still able to discover many XSS vulnerabilities. This indicates an opportunity for a more detailed analysis, by using a more thorough static analysis and interaction system.

The vulnerability class that FlashOver exposes, is still of interest today. Other researchers are now also working towards solutions to help prevent the exploitation of XSS vulnerabilities through Flash applications.

Flashbang [Cur] is an online service built around a modified version of Shumway [Moze], a free Flash emulator, allowing dynamic analysis of Flash applications without the need to decompile them. Just like FlashOver, Flashbang recognizes that the "FlashVars" parameters passed to Flash applications can result in security problems without proper input validation inside the Flash application. Flashbang locates variables that can be used through "FlashVars" and reports them so that their security impact can be analyzed without having to decompile the SWF file.

Phung et al. implemented FlashJaX [PMS+14], a fine-grained policy enforcement system that operates between Flash applications and the JavaScript environment. As FlashOver's results point out, untrusted JavaScript can leak into a web page's JavaScript through a Flash application, allowing an attacker to inflict damage. FlashJaX allows a web developer to safely integrate Flash applications without having to worry about leaks, by specifying a policy that restricts the Flash application's access to resources inside the JavaScript environment.

The standardization of HTML 5 brought new media functionality to modern browsers, making most common uses of Flash applications obsolete. Adobe, the developer of Flash, has stated that it plans to contribute more to HTML 5 instead of its own browser plugin [Adog].

The problem with Flash applications highlighted by FlashOver, is not restricted to the Flash plugin. Other browser plugins, such as Silverlight [micg], Java applets, PDF viewers, ... can potentially allow access to a web page's JavaScript environment. Vulnerabilities similar to those found in Flash applications, can thus also re-appear in other rich Internet applications. Developers should be aware of these extra conduits allowing untrusted JavaScript to leak into the JavaScript environment, so that they can properly validate input and prevent XSS vulnerabilities.

Abstract

Today's Internet is teeming with dynamic web applications visited by numerous Internet users. During their visits, typical Web users will unknowingly use tens of Rich Internet Applications like Flash banners or media players. For HTML-based web applications, it is well-known that Cross-site Scripting (XSS) vulnerabilities can be exploited to steal credentials or otherwise wreak havoc, and there is a lot of research into solving this problem. An aspect of this problem that seems to have been mostly overlooked by the academic community, is that XSS vulnerabilities also exist in Adobe Flash applications, and are actually easier to exploit because they do not require an enclosing HTML ecosystem.

In this chapter we present FlashOver, a system to automatically scan Rich Internet Applications for XSS vulnerabilities by using a combination of static and dynamic code analysis that reports no false positives. FlashOver was used in a large-scale experiment to analyze Flash applications found on the top 1,000 Internet sites, exposing XSS vulnerabilities that could compromise 64 of those sites, of which six are in the top 50.

3.1 Introduction

The last fifteen years have transformed the Web in ways that would seem unimaginable to anyone of the "few" Internet users of the year 1995 [Sto95]. What began as a simple set of protocols and mechanisms facilitating the exchange of static documents between remote computers is now an everyday part of billions' of users life, technical and non-technical alike. The sum of a user's daily experience is composed of open standards, such as HTML, JavaScript and Cascading Style Sheets as well as proprietary plugins, such as Adobe's Flash [adof] and Microsoft's Silverlight [micg].

Adobe's Flash is the most common way of delivering Rich Internet Applications to desktop users, with the latest statistics revealing an almost complete market penetration of Flash on desktop computers [fla, ria]. While some have claimed that the new version of HTML, HTML5 [htm], contains enough functionality to render the use of Flash obsolete, the reality is that today most Rich Internet Content, ranging from advertising banners and video players to interactive photo galleries and online games, is served and consumed by the Flash platform.

This rapid evolution of the Web was not left unnoticed by attackers. Traditionally, attackers preferred attacking the server-side of the Internet infrastructure, such as Web servers [Jk] and mail servers, since that gave

them access to powerful hosts with plenty of bandwidth and disk space as well as a foothold in a company's internal network. Nowadays however, the attacks are targeting the client-side of the Internet infrastructure. This can be the Web application, as rendered in a browser, the software of the browser itself or even the user sitting behind the browser. The result of client-side attacks is usually the theft of user credentials or the download of malware that makes the user's computer an unwilling part of a botnet [EWKK09].

Since Flash is part of all the technologies that shape the every day experience of Web users, it is also part of this new attack surface. Attacks against Flash target either vulnerabilities in the code of the Flash platform itself, or the insecure practices of developers of Flash applications. In this second category falls the problem of Cross-site Scripting (XSS) [xss]. While XSS in Web applications is a well-known and extensively researched problem, the problem of performing Cross-site Scripting attacks through vulnerable Flash applications has received much less attention. A Flash application can interact with the DOM (Document Object Model) of the page that embeds it or even with the browser itself. This allows Flash developers to read information from the page that embeds them, write information to the DOM or redirect the user to a desired page, such as the redirection that happens when a user clicks on a Flash advertisement banner. If these interactions are not protected adequately, an attacker can inject arbitrary JavaScript code that will be executed by a victim's browser in the context of the website hosting the vulnerable Flash application. Such code can, among others, steal a user's session identifier, access the website's local storage on a victim's browser or, in some cases, read the victim's geolocation information.

In this chapter we present FlashOver[1], a system capable of automated detection of Cross-site Scripting vulnerabilities in Flash applications. As the name of our system implies, its goal is to discover ways to perform malicious interactions between a Flash application and the rendering browser, that were never intended by the programmer of the vulnerable application. Given a Flash application, FlashOver performs static analysis in order to automatically identify ActionScript variables that can be initialized with user-input and are also used in operations that are commonly prone to code injection attacks. The identified variables are then tested dynamically in order to discover actual vulnerabilities present in the audited Flash application.

More specifically, our FlashOver prototype first decompiles the byte-code representation of ActionScript (the scripting language of the Flash platform) and then performs static analysis on the source code of the application, in search for commonly misused function calls that are responsible for *Flash-to-DOM* and *Flash-to-Browser* communication. Once these functions are located,

[1] *flashover*: An unintended electric arc, as between two pieces of apparatus

our system then tracks the arguments of these function calls back to their initialization. When this process is complete, the static-analysis component FlashOver produces a list of variables which are utilized in commonly misused ActionScript API calls and are initialized using user-input. This list of *potentially exploitable variables* is then used by the dynamic-analysis component of our system, which renders the Flash application in the Firefox browser and initializes the variables in many possible ways, always mimicking the methodology of attackers who would lure victims in a page under their control. In the last phase, the automatic clicking module of FlashOver clicks thousands of times on the rendered application, with the intent of triggering the vulnerable API call. If our system detects the execution of the injected JavaScript, then the Flash application is flagged as vulnerable.

To evaluate FlashOver, we obtained a partial list of Flash applications hosted on the top 1,000 sites of the Internet, which we downloaded and provided as input to our system. At the end of the experiment, FlashOver successfully detected exploitable XSS vulnerabilities in Flash applications of many well-known websites, including ebay.com, skype.com, mozilla.org and apple.com. These results are evidence both of the problem of XSS attacks through Flash applications as well as our system's ability of automatically detecting them. The main contributions of this chapter are the following:

- Detailed analysis of an XSS attack vector that is commonly overlooked in Web application development
- Design and implementation of FlashOver, a fully automated system which uses a combination of static and dynamic analysis in order to identify Flash applications vulnerable to code injection attacks
- Evaluation of our system using Flash applications of the top Internet websites, showing the prevalence of the aforementioned vulnerability as well as our system's ability of detecting it

The rest of this chapter is structured as follows: In Section 3.2 we give a brief overview of Cross-site Scripting attacks, Flash technology and how the one affects the other. We describe the general architecture of FlashOver in Section 3.3 followed by our implementation choices and their rationale in Section 3.4. In Section 3.5 we evaluate our prototype by using it to discover previously unreported vulnerabilities in Flash applications of the top 1,000 Alexa sites. We present our ethical considerations in Section 3.6, we discuss related work in Section 3.7 and we conclude in Section 3.8.

3.2 Background

In this section we give a brief overview of Cross-site Scripting attacks and of the Adobe Flash platform. We also present a motivating example showing how a vulnerable Flash application can be used to inject malicious JavaScript that will be executed by user's browser in the context of the domain hosting the vulnerable Flash application. While the techniques presented in the rest of this chapter are specific to the Flash platform, they are, in principle, applicable to other similar content-delivering platforms, such as Microsoft Silverlight [Mici].

3.2.1 Cross-site Scripting

Cross-site Scripting (XSS) attacks belong to a broader range of attacks, collectively known as code injection attacks. In code injection attacks, the attacker inputs data that is later on perceived as code and executed by the running application.

In XSS attacks, an attacker adds malicious JavaScript code on a page of a vulnerable website that will be executed by a victim's browser when that vulnerable page is visited. Malicious JavaScript running in the victim's browser and in the context of the vulnerable website can access, among others, the session cookies of that website and transfer them to an attacker-controlled server. The attacker can then replay these sessions to the vulnerable website effectively authenticating himself as the victim. The injected JavaScript can also be used to alter the page's appearance to perform phishing or steal sensitive input as it is typed-in by the user.

3.2.2 Adobe Flash

Adobe Flash is a proprietary multimedia platform which is used to create Rich Internet Applications. To be able to run Flash applications on a desktop, a Flash player must be installed which takes the form of a browser plugin. According to the latest statistics, Adobe's Flash player is installed on more than 99% of desktops connected to the Internet [fla, ria]. Over the years, the amount of functionality available to Flash applications has increased with each new version of the Flash player. Today, a Flash application can combine audio, video, images and other multimedia elements.

Flash applications are contained in SWF files (i.e. files with the .swf extension) which bundle multimedia elements together with byte-code-compiled ActionScript (AS) code. When loaded into the Flash player, the Flash

```
<object type="application/x-shockwave-flash" data="myFlashMovie.swf"
                        width="550" height="400">
        <param name="movie" value="myFlashMovie.swf" />
        <param name=FlashVars value="var1=Hello&var2=World" />
</object>
```

Listing 3.1: Embedding an SWF file using the object tag.

application is rendered and, if present, the AS byte-code is interpreted and executed. ActionScript is a scripting language developed by Adobe which allows the programmer to handle events, design the interaction between multimedia elements and communicate with both the embedding browser and remote Web servers. The current version of ActionScript is ActionScript 3.0 with legacy support for prior versions.

3.2.3 Using SWF files

SWF files are typically embedded in HTML using the `<object>` or `<embed>` tags, but it is also possible to load an SWF file into the browser directly, without embedding it into HTML, either by requesting it as is from a browser's URL bar or providing it as the source argument to an `<iframe>` tag in an existing HTML page.

Flash, like many other technologies, allows for the provision of load-time input next to hard-coded values specified at compile-time and present in the resulting SWF file. For instance, YouTube videos are displayed on webpages that each embed the same Flash video player. Data specific to the displayed video-file is passed to the Flash player at load-time through variables embedded in the enclosing HTML page. Flash supports two methods of passing values to Flash objects:

- **FlashVars directive**: When embedding an SWF file using the `<object>` or `<embed>` tags, the FlashVars parameter can be used to pass values to specific variables. In Listing 3.1, FlashVars are utilized to initialize Flash's variables `var1` and `var2` to "Hello" and "World" respectively.
- **GET parameters**: A web developer can also utilize GET-parameters to pass arguments to a Flash application. For instance, when the URI: `http://example.com/myFlashMovie.swf?var1=Hello&var2=`

`World` is invoked, the Flash application will initialize its internal variables `var1` and `var2` with their respective values. This method is usually overlooked by web developers who believe that the Flash application hosted on their page can only receive the parameters that they have hard-coded in the embedding HTML page and thus in many cases do not perform input validation within the Flash application itself.

3.2.4 Execution context of SWF files

In the previous section, we briefly examined the two ways that an SWF file can be loaded by a browser (using special HTML tags or a direct reference). While in both cases, the Flash Player loads the SWF file and starts executing it, there is a very important difference in the way that the two Flash applications interact with the surrounding page when the Flash applications requests the execution of JavaScript code from the browser.

The *allowScriptAccess* [Adob] runtime parameter arbitrates the access a Flash application has to the embedding page. There are three possible values: 'always', 'sameDomain' and 'never', with 'sameDomain' being the default. This value has the effect that access is only allowed when both the SWF application and the embedding page are from the same domain.

When an SWF file is embedded using the embed tag, and Flash requests the execution of JavaScript code from the browser, the code will execute within the origin of the embedding site, assuming a suitable value for the *allowScriptAccess* parameter. That is, if an SWF file hosted on the web server of `foo.com` is embedded in an HTML page on `bar.com`, the *origin* of the Flash-originating JavaScript is now `bar.com`. The origin is defined using the domain name, application layer protocol, and port number of the HTML document embedding the SWF.

If however, `bar.com` loads the SWF file of `foo.com` using an `<iframe>`, the browser creates an empty HTML page around the Flash application and any JavaScript initiated from the application will retain the origin of `foo.com`. Additionally, since the default value for *allowScriptAccess* is 'sameDomain', this means that the Flash application will be able to access data in the same origin as `foo.com`.

3.2.5 XSS in Flash

Consider a Flash advertising banner of which the ActionScript 2.0 source code is listed in Listing 3.2. The banner includes a button which, when

```
movie 'ad.swf' {
  button 42 {
    on (release) {
      getURL(_root.clickTag, '_blank');
    }
  }
}
```

Listing 3.2: ActionScript 2.0 source code of an example vulnerable Flash application.

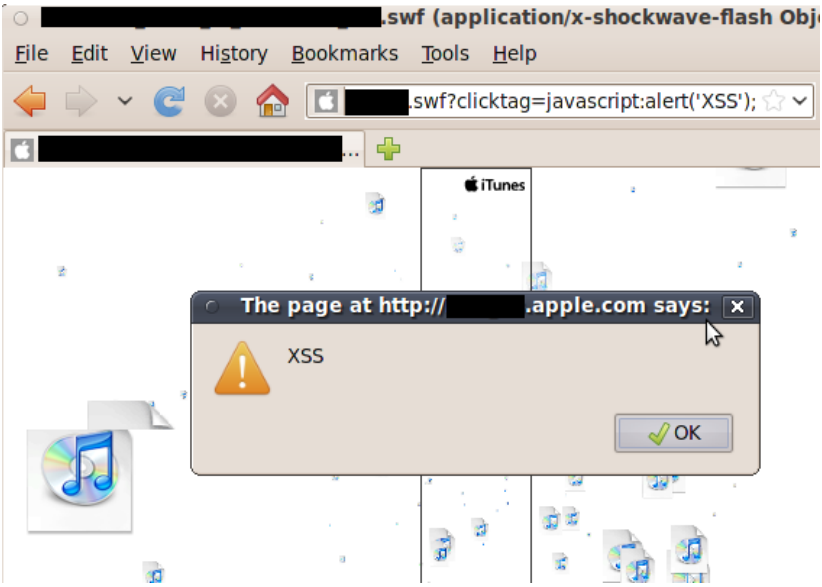


Figure 3.1: Advertising Banner on apple.com vulnerable to Cross-site Scripting through Flash.

clicked and released, triggers the execution of the `getURL()` function. The `getURL(url, target)` directs the browser to load a URL in the given target window. In this example, the URL is obtained from the variable `clickTag` in the global scope, and loaded into a new window (`_blank`). When used legitimately, the banner is located on `http://company.com/ad.swf` and is embedded on one of `company.com`'s web pages. The value of the `clickTag` variable is provided by the embedding page using the `FlashVars` directive and, in our example, suppose that it would redirect the clicking user to e.g.

http://company.com/new_product.html.

As described in earlier sections, an SWF file can be directly referenced and any GET parameters will be provided to the Flash application itself, exactly as in the `FlashVars` case. Thus, if the banner was directly requested through `http://company.com/ad.swf?clickTag=http://www.evil.com`, the `clickTag` variable would now hold the value `http://www.evil.com` instead of the value intended by `company.com`. This behavior could be abused by attackers in order to send malicious requests with the correct Referrer header towards Web applications that use Referrer checking as a means of protection against CSRF attacks [Shi]. While this is definitely a misuse scenario, the vulnerable code unfortunately allows for a much greater abuse. Instead of providing a website URL as the value for `clickTag`, an attacker could provide a JavaScript URL, such as `javascript:alert('XSS')`. A JavaScript URL is a URL that causes the browser to execute the specified JavaScript code in the context of the current-page (`alert('XSS')` in our aforementioned example) instead of making a remote request, as is the case in HTTP(S) URLs. In this scenario, when that banner is clicked, the user's browser will execute attacker-supplied JavaScript code instead of redirecting the user.

All an attacker needs to do in order to exploit this vulnerability, is to lure a victim into visiting a website which loads the vulnerable SWF file in an `iframe` and insert a `javascript:` URL containing malicious JavaScript code into the query string of the SWF file URL. Since the SWF file is loaded in an `iframe`, it will retain the origin of `company.com` and thus when the user clicks on the banner, the JavaScript code will execute in the context of `company.com` instead of the attacker's site. This will allow the malicious JavaScript code to access, among other things, the user's cookies for `company.com` and steal his session identifiers. If a click on the vulnerable Flash banner is required to trigger the execution of the injected JavaScript, the user can be tricked into clicking the banner, either using social engineering or clickjacking techniques [BEK+10]. In cases where the vulnerable code is triggered after a predetermined amount of time, all that the attacker needs to do is to make sure to keep the user on his malicious site for the appropriate amount of time.

While the example ActionScript in Listing 3.2 appears to be a contrived one, many websites unfortunately have similarly vulnerable banners. Figure 3.1 shows a banner hosted on `apple.com`[2] which does not perform input validation within its ActionScript code and is thus vulnerable to XSS.

[2]We discovered this vulnerable SWF file through our experiment described in Section 3.5, and we also responsibly informed Apple about this vulnerability, see Section 3.5.3

3.3 FlashOver approach

The goal of FlashOver is to automatically discover XSS vulnerabilities in Flash applications, as opposed to the manual code review illustrated in Section 3.2.5. Logically, FlashOver can be separated in three sequential steps: static analysis, attack URL construction and automated interaction. The high-level idea behind each of these steps of this approach is explained in more detail in the following subsections.

3.3.1 Static analysis

In this first step, *potentially exploitable variables* (PEVs) are automatically discovered in a given SWF file. PEVs are variables which are utilized in commonly misused ActionScript API calls and are initialized using user-input. This step requires a static analysis of the ActionScript byte-code embedded in the given SWF file.

Embedded ActionScript byte-code in an SWF file cannot easily be read and understood by a human, giving a false sense of security to Flash developers who think their code cannot be recovered. In reality, several free and commercial SWF decompilers exist that can reconstruct the ActionScript source code with very high accuracy.

Be it either through decompilation and source code analysis, or static analysis of the ActionScript byte-code, a list of potentially exploitable variables is extracted from the SWF file. The variables in this list will be used as attack vectors in later steps of FlashOver.

3.3.2 Attack URL construction

In this second step, an actual attack on the Flash application is prepared by crafting the *attack URL* that an attacker would give to a victim and trick him into navigating to it. In an actual XSS attack the attacker would try to execute JavaScript in the security context of a target domain using the victim's credentials for that domain. While the attacker's injected JavaScript would perform something undesirable for the victim, FlashOver uses the injected JavaScript code to log that the attack was successful.

The results of FlashOver will ultimately be used by Flash application developers to track down vulnerabilities in their code and fix them. Therefore it is essential that the results provide as much useful data as possible. There are three

essential pieces of information that must be recorded to be able to reconstruct a successful attack: the *entry point* (i.e. Flash application that was exploited), the *attack vector* (i.e. the exploitable variable used to inject code) and the *payload* (i.e. the injected JavaScript code).

These three pieces of information are encoded in the attack URL. The SWF file being attacked can be identified by a unique identifier `swfid`. For each variable `var` of the potentially exploitable variables, as identified in the static analysis step, a payload value of payload-type `type` is generated. This payload contains JavaScript code that, when executed by the targeted Flash application, will log the tuple `(swfid, var, type)`. From any tuple `(swfid, var, type)` that shows up in the logs, the entry point, attack vector and payload can be reconstructed and can be used to identify the exact vulnerability of the Flash application.

3.3.3 Automated interaction

In the third step of the FlashOver process, the previously crafted attack URLs are used to truly attack the Flash application being examined. In a real-world scenario, the attacker would give the attack URL to a victim and trick the victim into interacting with the given Flash application. Since FlashOver tries to match the scenario as close to reality as possible, an automated process must interact with the Flash application and by doing so, trigger the execution of the JavaScript payload encoded in the attack URL.

Interaction can mean a lot of things. Flash applications can respond to keyboard events, mouse events and even more esoteric events from e.g. a built-in tilt sensor. The set of input events that trigger actions in a Flash application depends on the Flash application itself. For good results, the automated interaction process should try to cover as much as possible in an intelligent way.

3.4 FlashOver Prototype

The description of the general FlashOver approach in Section 3.3 omits implementation details, because each of the steps in FlashOver can be implemented in a number of ways with varying degrees of thoroughness. We purposefully chose to implement a minimalistic version of FlashOver to investigate the level of effort and skill required by an attacker to automatically detect XSS vulnerabilities in SWF files.

Our FlashOver prototype is schematically illustrated in Figure 3.2. The following subsections discuss the implementation details of each step in our FlashOver prototype.

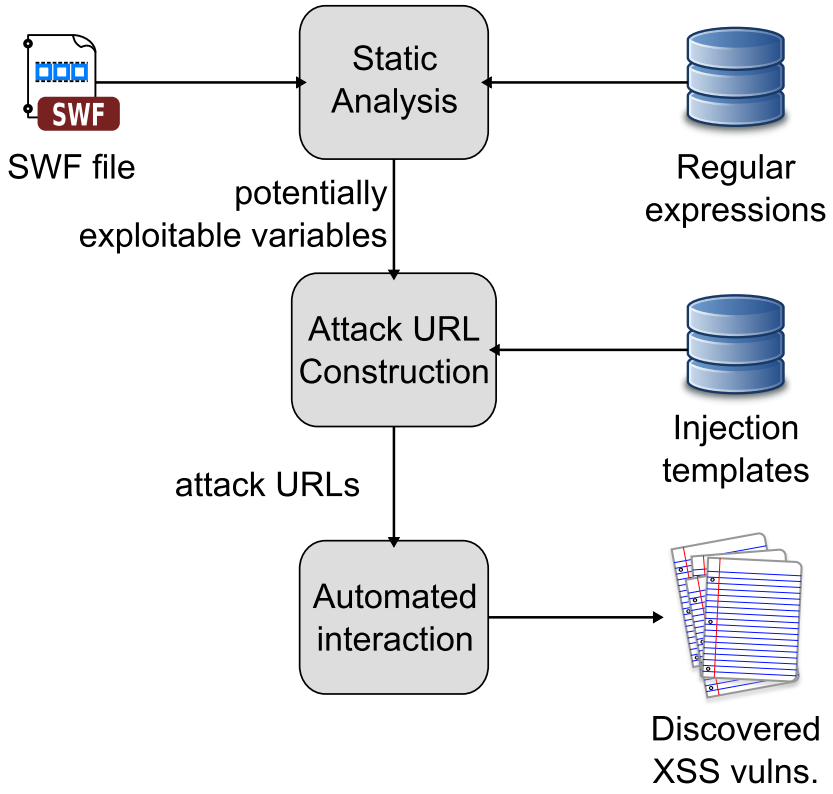


Figure 3.2: Schematic overview of our FlashOver prototype: During static analysis, the SWF file is decompiled and regular expressions uncover *potentially exploitable variables* (PEVs) from the ActionScript source-code. These PEVs are inserted into injection templates in the attack URL construction step. The attack URLs are loaded in a real browser in the automated interaction step, resulting in a list of discovered XSS vulnerabilities.

3.4.1 Static analysis

This first step in the FlashOver process requires static analysis of the SWF file. We chose to decompile the SWF file and then perform a simple static analysis on the resulting ActionScript source code.

There are many SWF decompilers, but not all of them support ActionScript 3.0. Choosing a decompiler, such as the freely available `flare` [Kog], that does not support the latest version of ActionScript, would mean that there would be a blind-spot in our analysis. For that reason, we chose a commercial decompiler with support for ActionScript 3.0 [sot].

To reduce the complexity of our prototype, we opted for a simple regular-expression extraction of the PEVs instead of using more complicated analysis methods. Using this method, the resulting ActionScript source code is searched for patterns indicating potentially exploitable variables.

- `_root.`*re*
- `getRemote(#,`*re*`,...)`
- `.addCallback(#,#,`*re*`)`
- `.sendAndload(`*re*`,...)`
- `loadvariables(`*re*`,...)`
- `URLRequest(`*re*`,...)`
- `getURL(`*re*`,...)`
- `loadMovie(`*re*`,...)`
- `.load(`*re*`,...)`
- `.call(`*re*`,...)`
- `loadClip(`*re*`,...)`

where the regular expression to match a variable name *re* = '`[a-zA-Z$_][a-zA-Z0-9$_]*`' and '#' denotes a "don't care" parameter.

Figure 3.3: The regular expressions, in pseudo-form, used in our FlashOver prototype to match the names of potentially exploitable variables.

The regular expressions used in our prototype are listed in pseudo-form in Figure 3.3. For each of these regular expressions, *re* indicates where the name of a potentially exploitable variable could appear in a function call in the ActionScript source code. The regular expression used to match variable names is synthesized from the variable naming rules defined by Adobe: *"The first character of an identifier must be a letter, underscore (_), or dollar sign ($). Each subsequent character can be a number, letter, underscore, or dollar sign"* [Adoa]. The first regular expression (`_root.`*re*) indicates that a variable in the global address space is used, while the other regular expressions match function calls for sensitive functions that could lead to XSS.

3.4.2 Attack URL construction

Based on the variable names identified in the previous step, attack URLs are constructed that, when the attack payload is triggered, will report in what way the given SWF file is vulnerable to XSS.

Table 3.1: The 10 injection templates used in our implementation. Each injection template matches a certain example occurrence of a exploitable variable in ActionScript. The injection template indicates what data should be injected for a successful attack. The first template is a control, where the logging URL is injected instead of any code. The other nine inject actual JavaScript code.

id	**Example occurrence of *var***	**Contents of *var***	
0	`getURL(`*var*`)`	*target URL*	control - plain target URL
1		`javascript:`*code*	JavaScript URL
2	`getURL("javascript:"+`*var*`)`	*code*	JavaScript code by itself
3	`writeHTML(`*var*`)`	`<script>`*code*`</script>`	HTML <script> tag injection
4	`eval("x = "+`*var*`+";")`	`0;` *code*`//`	introducing closing quotes and semicolons
5	`eval("x = '"+`*var*`+"';")`	`';` *code*`//`	
6	`eval("x = \""+`*var*`+"\";")`	`";` *code*`//`	
7	`eval("alert("+`*var*`+")")`	`0);` *code*`//`	introducing closing quotes, brackets and semicolons
8	`eval("alert('abc = '"+`*var*`+"')")`	`');` *code*`//`	
9	`eval("alert(\"abc = \""+`*var*`+"\")")`	`");` *code*`//`	

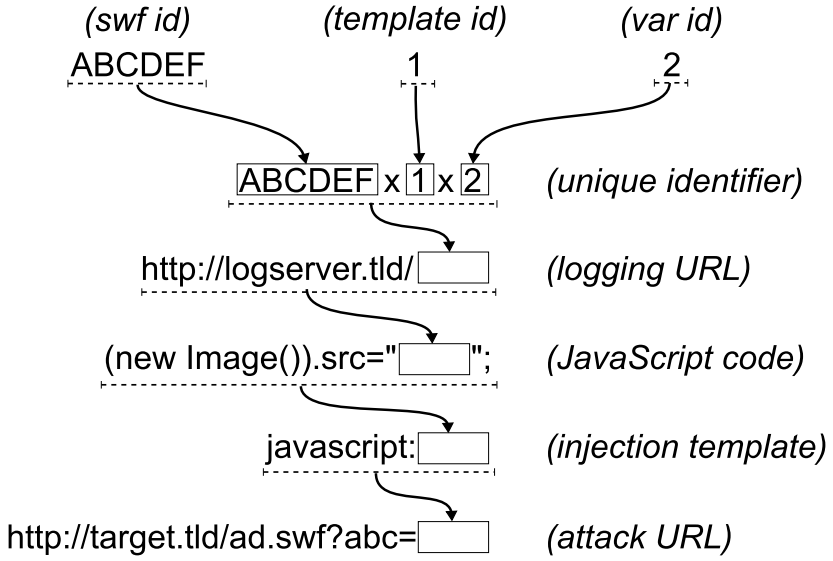


Figure 3.4: Construction process of an attack URL for `http://target.tld/ad.swf` with swfid `ABCDEF`, injection template id `1` and variable `abc` with id `2`.

Exploitable variables can be used in ActionScript in a number of different ways. Through our review of JavaScript injection techniques, we identified a non-exhaustive list of nine ways in which an attacker-specified payload can ultimately be injected into a JavaScript context, through exploitable variables in an SWF file. As a control, we also use an injection template that injects no JavaScript code. The injection templates are summarized in Table 3.1. For each of these injection templates, a separate attack URL is constructed.

As discussed in Section 3.3.2, the attack URL should encode information about entry point, attack vector and payload type into a unique identifier. The entry point is encoded by a unique hex-encoded 256-bit number that identifies the SWF file being analyzed. The attack vector, or the exploitable variable used to inject the payload, is encoded as an index into the list of identified potentially exploitable variables. Finally, the payload type is encoded as an index into the list of nine injection templates specified earlier.

The process for building an attack URL for an example SWF file with `swfid` equal to `ABCDEF`, an exploitable variable `abc` and injection template `1` is shown in Figure 3.4. From the given SWF file identifier (swfid), injection template index (type id) and exploitable variable index (var id), a unique identifier is

constructed for this specific attack URL, by concatenating these three values, separated by a 'x' character. This unique identifier is appended to the URL for the log-server, forming the logging URL. The logging URL is then used in a JavaScript code fragment that, when executed, will trigger a request to the log-server, logging the unique identifier. This piece of JavaScript code is then inserted into the selected injection template, forming the payload of the attack URL, in this case a simple `javascript:` URL. Finally, the payload is assigned to the exploitable variable (`abc` in Figure 3.4) in a query string of the attack URL.

3.4.3 Automated interaction

The final step of FlashOver, involves passing the crafted attack URL to a simulated victim and let that victim interact with it, potentially triggering the execution of the injected JavaScript. Based on our personal experience and the analysis of many Flash applications, we make the assumption that most interactions with Flash applications are achieved through mouse clicks. For that reason, we only consider this type of interaction in our prototype implementation.

The Flash application is loaded into a real Firefox browser. The browser itself is started in `Xvfb`, a virtual frame-buffer X server [3] and the virtual mouse attached to this `Xvfb` session is controlled through the `xte` program [4]. The `Xvfb` server is set up to offer a virtual frame-buffer of 640x480 pixels with 24-bit color to any program running inside. Firefox, running inside `Xvfb` is started full-screen (so 640x480) in kiosk mode. This means that all toolbars and menus are removed, and undesirable functionality, like printing, is disabled.

Once Firefox has started and loaded the Flash application, a list with 10,000 random (x,y) locations is generated and passed to `xte`, which moves the mouse to those locations and issues a click. After these 10,000 clicks, the automated clicker pauses to give the Flash application time to process the input, which could involve executing the injected JavaScript payload.

If the execution of the injected JavaScript is triggered as a result of one or more mouse-clicks, this will be recorded in our logging server. The detection of the injected codes' execution effectively creates a new set of *actually exploitable variables* which is a subset of the original *potentially exploitable variables* set, as that was generated in the first stage of FlashOver. The entries of the logging server can then be used, as previously explained, to pinpoint the exact place in

[3] http://www.xfree86.org/4.0.1/Xvfb.1.html
[4] http://linux.die.net/man/1/xte

the Flash application and the specific attack vector that can be used for a XSS attack.

3.5 Evaluation

We evaluated our FlashOver prototype with a large-scale experiment to determine how many SWF files vulnerable to XSS are hosted on the Alexa top 1,000 Internet sites [ale].

3.5.1 Experimental setup

For each of the domains in the Alexa top 1k, a list of publicly exposed SWF files was retrieved from Altavista using the query "*site:domain.com filetype:swf*" where domain.com would be a domain in our experiment.

The SWF files discovered through these queries were downloaded onto a local web server. Although the experiment could have been conducted using the SWF hosted on their original locations, we feared that it might potentially harm the targeted site. In addition, storing the SWF locally improved performance by reducing the time it took to load the SWF file into the browser.

After the non-SWF or otherwise invalid SWF files were removed from the set of downloaded files, they were processed by FlashOver. The static analysis and attack URL construction steps of FlashOver were performed on all SWF files in advance to reduce overhead for the entire experiment. The final step, using an automated clicker, was performed in parallel on 70 dual-core computers.

Because the automated clicker clicks on random positions on the Flash application, each run of the automated clicker can yield different results. To increase the odds that the payload in the attack URLs was triggered, the entire dataset was processed by the automated clickers 20 times. The total experiment ran for approximately five days, approximately six hours per run.

3.5.2 Results

From Altavista, 18,732 URLs were retrieved. After downloading, 3,800 SWF files did not contain a valid Flash application. Of the remaining 14,932 SWF files, 35 caused our decompiler to destabilize and crash. From the 14,897 SWF files that were decompiled successfully, 8,441 were determined to have exploitable variables. For each of these 8,441 SWF files, 10 attack URLs were generated:

one for each injection template listed in Table 3.1. The final generated dataset contained a list of 84,410 attack URLs. All of these were processed in parallel by the automated clickers.

After analysis of the log files, 523 SWF files were found to load content from an attacker-supplied URL (i.e. *URL injection*) and 286 SWF files allowed the execution of attacker-supplied JavaScript code. These 286 vulnerable SWF files can be traced back to 64 Alexa domains, of which six are in the top 50.

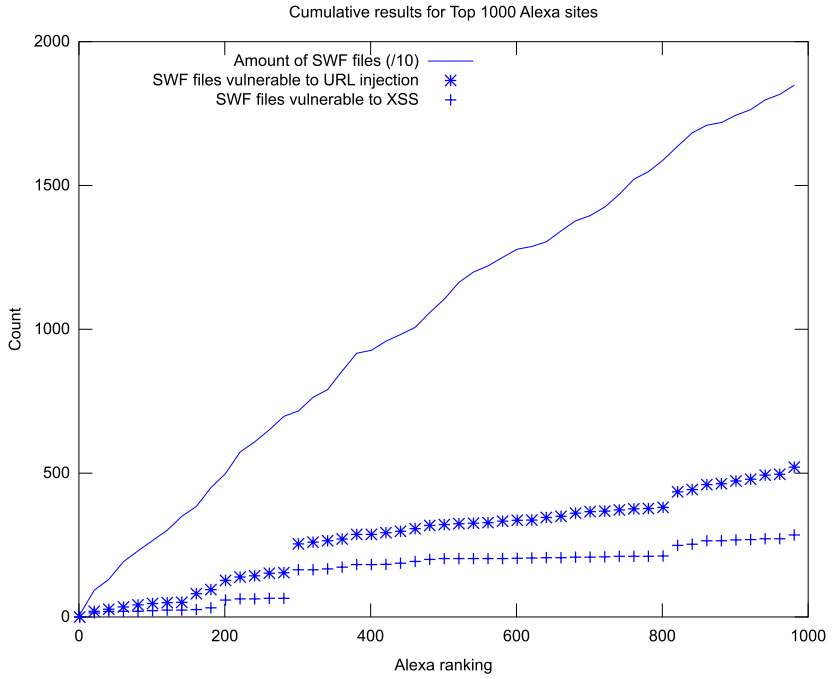


Figure 3.5: Results from our FlashOver experiment, shown as a cumulative plot. The amount of SWF files per site found is divided by 10 to match the scale of the other results.

The results of our large-scale experiment are summarized in the cumulative plot in Figure 3.5. The data-points are sorted on the x-axis, lower values indicating higher Alexa ranking, and vice versa. Three data-points per Alexa domain are shown: the amount of SWF files found per domain, divided by 10 to match scale, the amount of SWF files in that domain vulnerable to URL injection and the amount of SWF files vulnerable to XSS. The three distinguishable jumps, at indices 193, 293 and 806, indicate a large amount of vulnerable SWF files located at the Alexa domains of the corresponding ranking.

Table 3.2: Top ten most commonly-named vulnerable variables found in our experiment.

Variable Name	Instances found	Percentage
clicktag	101	35.31%
pageurl	97	33.92%
click	26	9.10%
counturl	10	3.50%
gameinfo	8	2.80%
link1	7	2.44%
url	3	1.05%
link04	2	0.70%
downloadaddress	2	0.70%

Table 3.2 shows the ten most commonly named vulnerable variables that we discovered in our analysis. Interestingly, the two most commonly vulnerable variables are responsible for more than 69% of all vulnerabilities found. The fact that many different Flash applications are vulnerable to the same attack and through the same variables, suggests the use of automated tools for the creation of Flash applications that generate code in a vulnerable way. At the same time, our results highlight the need for scanning of variables and code-paths beyond the ones commonly associated with vulnerabilities.

3.5.3 Discussion

When one considers the number of vulnerable Flash applications found on the Internet's top websites, it becomes clear that XSS attacks through Flash applications are indeed a problem. Although Adobe advocates security best practices [Adod], stating that user-input should be sanitized where needed, this advice seems to be overlooked by Flash application developers.

The required effort and skill to automatically discover these XSS vulnerabilities is limited. As discussed in Section 3.4, our FlashOver prototype uses suboptimal static analysis and randomized clicking to simulate a user. For the static analysis part, a more precise taint-analysis system would produce better results since it could identify more variables influenced by user-input and thus produce a longer list of *potentially exploitable variables*. Moreover, a determined attacker can easily uncover additional vulnerabilities using a manual static analysis. Likewise, the randomized clicker is lacking the cognitive ability of an actual

human user: it does not understand typical GUI widgets that a human would click and it cannot interact with e.g. a game like a human would. This means that there may be vulnerabilities that our clickers could not trigger but that a human victim would. Therefore, the amount of vulnerable Flash applications detected in this experiment is a lower bound: the actual amount of vulnerable applications is most likely higher, making the security threat an even bigger issue.

An interesting property of FlashOver is that it detects successful JavaScript injection by actually simulating a victim who triggers the use of the injected JavaScript code in one or more potentially exploitable variables. Thus, while FlashOver may miss some vulnerabilities (false negatives), it has practically zero false positives. While one can construct examples where FlashOver would report a false positive, e.g. an application that is vulnerable to XSS but inspects the injected payload and only allows it if it is "not dangerous", we believe that these are unrealistic examples and thus would not be encountered in the analysis of real-life Flash applications.

3.6 Ethical Considerations

Testing the security of real websites against Cross-site Scripting attacks may raise some ethical concerns. However, analogous to the real-world experiments conducted by Jakobsson et al. [JFJ08, JR06] and Nikiforakis et al. [NBVA+11], we believe that realistic experiments are the only way to reliably estimate success rates of attacks in the real world. Moreover, we believe that our experiments will help raise awareness against this, usually overlooked, issue. In particular, note that:

- All Flash applications were downloaded and exploited locally thus no malicious traffic was sent towards the live Web servers of each website
- All attacks were targeting our own simulated victim and no real users
- We are in the process of disclosing these vulnerabilities to all the affected websites so that they may repair them

3.7 Related work

Due to the large installation percentage of Adobe's Flash in desktop and laptop computers, Flash has been the target of many attacks over the years. These

attacks have been targeting either implementation bugs in the Flash plugin itself [Bla10] or the insecure use of Flash functionality from Rich-Internet Application developers.

Cross-site Scripting attacks in Web applications [xss] have received a lot of attention over the last years and there exists a wide range of research on detecting injected JavaScript and protecting the user from it [KKVJ06, VNJ+07, NMY+11, VGC09] as well as many initiatives that try to educate developers about this issue [owac, Con]. The sheer volume of XSS attacks has even caused mainstream browsers like Microsoft Internet Explorer 8 and Google Chrome to add XSS-detection mechanisms in an attempt to stop attacks against the browsing user, even if the visited Web application is not actively protecting itself [Bar, Micb].

The problem of performing Cross-site Scripting attacks through insecure Flash API methods was first highlighted by Jagdale [Jag09] who provided examples of insecure ActionScript code and reported that out of the first 200 SWF files that Google gave as a result to the search query "`filetype: swf inurl:clickTag`", 120 were vulnerable. Jagdale also showed that many tools that automatically generated SWFs were, at the time, generating applications vulnerable to XSS attacks, including tools by Adobe itself. Bailey [Bai10] verified the earlier findings of Jagdale and gave examples of high-profile websites hosting SWFs vulnerable to Remote File Inclusion attacks (RFI) that could be leveraged to perform, among others, XSS attacks.

SWFScan [Hew] is a tool that decompiles a Flash application and performs static analysis to detect possible vulnerabilities. SWFScan searches a decompiled Flash application for hardcoded URLs, passwords, insecure cross-domain permissions and coding practices that may lead to XSS. SWFIntruder [Ste] is a user-guided semi-automatic tool which tests for XSS vulnerabilities in Flash applications.

The important difference that separates FlashOver from earlier work is that earlier work depended either on the manual or semi-automatic analysis of SWF files. Contrastingly, FlashOver is the first system that is able to discover "zero-day" vulnerabilities in a completely automatic fashion without relying on naming conventions of commonly vulnerable variables or user guidance. While FlashOver, due to its incomplete static analysis, may miss some vulnerabilities (false-negatives), it produces no false-positives since any reported vulnerability could only have been reported because that vulnerability was exploited.

Another problem that has attracted attention from the security community is the existence of insecure cross-domain Flash policies. The Flash plugin is able to conduct Cross-Domain requests in a way that violates the Same-Origin policy that exists in JavaScript. In order to overcome this problem, any website that

wants to be contacted through Flash, must opt-in by placing a cross-domain policy file in its root directory that specifies which domains can be accessed and in what ways. Three recent independent studies [KAPM11, JVSS11, LJT11] all discovered that a great number of websites deploy insecure cross-domain policies in a way that allows their users to fall victims to impersonation attacks, simply by browsing to a malicious website.

An interesting observation is that over the last few years, many researchers have shifted their focus and have designed and implemented a number of *blackbox* and *whitebox* systems that, like FlashOver, attempt to automatically detect vulnerabilities in Web applications. These systems are usually less precise than human analysts but can process data much faster and can track dependencies among hundreds of files. Balduzzi et al. [BEK+10] presented a system that automatically discovers clickjacking attacks through an instrumented Firefox browser and a series of plugins that detect the overlay of many objects at specifics coordinates within a Web page. NoTamper, by Bisht et al. [BHS+10], detects vulnerabilities that would allow a user to successfully perform HTTP parameter-tampering. Ford et al. [FCKV09] propose OdoSwiff, a system to detect deliberately malicious Flash ads through a combination of static and dynamic analysis.

Jovanovic et al. [JKK06], Xie et al. [XA06] and Wassermann et al. [WS07] use static analysis on a Web page's source code in an effort to identify potential flaws that could lead to XSS, SQL injections and command injection attacks. Sun et al. [SXS11] use static analysis to infer the intended access-control of Web applications and use their models to detect access control errors.

3.8 Conclusion

The constant innovation in the World Wide Web has allowed developers to use more and more the browser as the platform of choice for delivering content-rich applications to users. In this picture, the Flash platform by Adobe plays a very important role and is widely used in modern websites. However, since Adobe is a Web technology, it is also part of the modern attack surface where the targets are now the users and their browsers. In this chapter, we analyzed the implications of making the wrong assumptions in the Flash platform and we presented FlashOver, the first fully automated discovery system for XSS attacks, specific to Flash. FlashOver uses a combination of static and dynamic analysis to identify vulnerabilities in real-life Flash objects and using our system, we discovered that a significant number of high-valued websites host Flash applications that are vulnerable to Cross-Site Scripting. These results

attest towards the importance of this attack vector and we hope that our work will help raise awareness of insecure coding practices in the community of Rich Internet Application developers.

Acknowledgments

We would like to thank our shepherd, Dieter Gollmann, and the anonymous reviewers for their insightful comments that helped to greatly improve the presentation of this chapter. This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, the IBBT, the Research Fund K.U.Leuven, the B-CCENTRE and the EU-funded FP7 projects NESSoS and WebSand.

Chapter 4

Monkey-in-the-Browser: Malware and Vulnerabilities in Augmented Browsing Script Markets

Publication data

Contained in this chapter is the paper "Monkey-in-the-browser: Malware and vulnerabilities in Augmented Browsing Script Markets," presented at the 9th ACM Symposium on Information, Computer and Communication Security (ASIACCS 2014) [AND+14, VND+14]. Steven Van Acker was the lead author of this work.

Preamble

This chapter reviews how attacker-controlled JavaScript code can be injected into web pages through browser extensions such as Greasemonkey.

This chapter studies Greasemonkey, a browser extension implementing an augmented browsing framework, and its script market userscripts.org.

Augmented browsing is the practice of modifying a web page in the client side browser to augment or restructure data on that web page. Greasemonkey is a popular browser extension, with more than 1.5 million users [gre], facilitating augmented browsing.

To use Greasemonkey, the user creates or downloads a "user script," a file consisting of JavaScript with some meta-data, and registers it with the Greasemonkey extension. The meta-data associated with the user script instructs Greasemonkey in which web pages the user script should be included through the @include header. When included, the user script executes its JavaScript code on the web page as if it were included by the web developer of that page.

Greasemonkey provides executing user scripts with access to some powerful functionality that is not limited by the same-origin policy. To avoid that this functionality leaks into the JavaScript execution context for all scripts to use, Greasemonkey isolates that functionality in a sandbox where the user script also runs. Afterwards, that sandbox is discarded.

Userscripts.org is the official community of user script developers that share their scripts. Unsuspecting users of this "script market" will typically install user scripts without reviewing the source code, opening the way for malware on this community website. A self-regulated malware detection process is in place on userscripts.org, allowing users to flag user scripts as malicious.

We reviewed the 86,358 user scripts on userscripts.org and found that the self-regulated malware review process has a very high false-positive ratio of almost 80%. Furthermore, we used Mjolnir, a framework for large-scale experimentation further described in Section 8.3, to analyze all the available user scripts on userscripts.org. We analyzed the metadata and performed static analysis the JavaScript code of all these user scripts and found that many of them contain DOM-XSS vulnerabilities and around 60% of them can be tricked into executing on unintended web pages through an overly generic `@include` header.

In addition, we have identified 58 user scripts which are so vulnerable that they can be tricked by an attacker to execute on any web page, executing attacker-controlled code in that web page's origin and have access to the very powerful same-origin-exempt functionality available in the Greasemonkey sandbox. Such user scripts are vulnerable to what we call a Privileged, Global XSS. The most prominent example of a script vulnerable to a privileged, global XSS, puts more than 1.2 million users at risk.

The main contributions of this research are:

- Analysis of security-sensitive functionality in the Greasemonkey extension,

- Analysis of the userscripts.org script market with regard to the self-regulated malware review process,
- Static analysis of the user scripts in this script market for DOM-XSS vulnerabilities,
- Demonstration of novel attacks which leverage functionality specific to Greasemonkey and benign user scripts into a powerful attack on secure websites.

In hindsight, this work was the first to analyze Greasemonkey's community-driven script market for XSS vulnerabilities. Our analysis shows what can happen when attacker-controlled JavaScript can leak into the JavaScript execution environment through a browser extension. The vulnerabilities we found are specific to Greasemonkey and its user scripts and cannot easily be extrapolated to other browser extensions. However, the gravity of the discovered vulnerabilities illustrates what the potential impact can be when attacker-controlled JavaScript can leak through a browser extension into a JavaScript execution environment.

Research by Barth et al. [BFSB10] found that 88% of the 25 most popular Firefox browser extensions do not use the full set of privileges available to them, and that 76% of them use unnecessarily powerful APIs. They propose a least privilege approach for browser extensions, to prevent that vulnerabilities in browser extensions can lead to the abuse of powerful functionality by an attacker. Our work is centered around Greasemonkey, which is itself a browser extension.

While we agree that a least privilege approach is a good idea for browser extensions, it would have a limited impact on vulnerabilities in user scripts used by Greasemonkey, because it requires the superset of all functionality required by its user scripts. A least privilege approach enforced on user scripts by Greasemonkey itself, would help prevent vulnerabilities such as privileged, global XSS. Approaches to isolate and restrict JavaScript code according to the least-privilege principle are discussed in Chapter 5.

Shortly after our work was published, the official userscripts.org community website was discontinued, and a copy of this website has appeared elsewhere [uso]. Similar script markets have also surfaced, but they do not yet address the shortcomings of userscripts.org's malware review process. Without better quality control, these new script markets will expose their users to the same security problems as those found by our research.

Abstract

With the constant migration of applications from the desktop to the web, power users have found ways of enhancing web applications, at the client-side, according to their needs.

In this chapter, we investigate this phenomenon by focusing on the popular Greasemonkey extension which enables users to write scripts that arbitrarily change the content of any page, allowing them to remove unwanted features from web applications, or add additional, desired features to them. The creation of script markets, on which these scripts are often shared, extends the standard web security model with two new actors, introducing newly identified types of vulnerabilities.

We describe the architecture of Greasemonkey and perform a large-scale analysis of the most popular, community-driven, script market for Greasemonkey. Through our analysis, we discover not only dozens of malicious scripts waiting to be installed by users, but thousands of benign scripts with vulnerabilities that could be abused by attackers. In 58 cases, the vulnerabilities are so severe, that they can be used to bypass the Same-Origin Policy of the user's browser and steal sensitive user-data from all sites.

We have discovered several of these severely vulnerable scripts, with over a million installations, and created a proof-of-concept exploit that successfully launches a novel "Man-in-the-browser" attack against an installed vulnerable script with an installation base of 1.2 million users.

4.1 Introduction

The web has evolved from a collection of purely static pages to entire web applications, making the browser the medium of choice for delivering new software and services. For many users, the desktop appears to do little more than house their browser and manage their Internet connection. With this migration, many power users who used to customize their operating system and install their applications of choice, now feel the desire to customize the applications inside their browser, in a way that fits their needs. These customizations usually result in an enhanced form of browsing the web, which is called "augmented browsing."

Probably the most well-known instance of augmented browsing software is the *Greasemonkey* [gre] browser extension, which, at the time of this writing, ranks fifth in the list of most popular Firefox extensions [top]. Greasemonkey

users can write *user scripts*, i.e., small JavaScript programs, that manipulate loaded webpages on the client-side in any way desired. User scripts can, among others, hide ads, change the content layout of a page, and make cross-origin HTTP requests to create client-side mashups. In contrast with typical browser extensions, user scripts are comprised of a single JavaScript file and are not packaged in any way, making them easy to inspect and modify. Overall, Greasemonkey and user scripts tend to a different audience than the neatly-packaged browser extensions available on the traditional extension markets.

Due to the popularity of Greasemonkey and the large number of user scripts created for it, the Greasemonkey developers created a community website on which members can exchange user scripts: a community-driven, script market known as `userscripts.org` [use].

The creation of a script market brings along some unique security issues, because it extends the standard web attacker model with new actors. In the regular model, a website is visited by a client and an attacker can either attack the website by exploiting server-side vulnerabilities, or the visitor through client-side vulnerabilities, like XSS or CSRF. In the augmented browsing scenario, however, the model is extended with the inclusion of a user script in the visitor's browser, a script market with user scripts, and a script author creating and sharing user scripts through the script market.

In this chapter, we perform an in-depth analysis of this extended script ecosystem. First, we consider the script author as a malicious actor, having the ability to create user scripts with malicious functionality, and upload them to the script market where they may be downloaded and installed by victim users. We report on the prevalence of malicious scripts, the discovered malice, and whether this malice was identified by the `userscripts.org` community.

Next, we briefly look at specific scenarios allowing targeted attacks against script users without their knowledge, either at script installation time or any other time during the lifetime of a script within their augmented browser.

Last, we shift our focus to the possibility of conducting attacks on poorly coded user scripts. We find many instances of benign scripts whose authors, even though they had no bad intentions, unwillingly introduced vulnerabilities which could be used to attack websites that are otherwise secure. Using straightforward static-analysis techniques, we identify more than 100 user scripts, with millions of installations, vulnerable to DOM-based XSS. We also show that a certain type of user script vulnerability can be abused to launch attacks even against the Greasemonkey engine itself, leading to powerful global XSS attacks, where an attacker can steal a user's data from all sites.

Our main contributions are:

- We evaluate the Greasemonkey browser extension, focusing on the functionality with negative security consequences.
- We analyze the most popular, community-driven script market for Greasemonkey and describe the difficulties of relying on the community to define and identify maliciousness.
- We demonstrate novel attacks that take advantage of benign Greasemonkey scripts to attack, otherwise secure, websites.

4.2 Greasemonkey

In this section, we describe the Greasemonkey engine, its uses, and the structure of Greasemonkey scripts. Finally we examine how Greasemonkey affects the security and isolation of scripts in the browser.

4.2.1 Greasemonkey engine

Greasemonkey is a popular browser add-on for augmented browsing. Using Greasemonkey, users can, on the client side, modify the appearance and functionality of any page of the web. This is done by JavaScript programs that are injected in arbitrary webpages and have access to privileged functionality, not available to normal JavaScript programs. Through these Greasemonkey scripts and with the help of the browser's DOM, users can arbitrarily edit a webpage, including the removal of content, e.g., ads, or the addition of new content, e.g., adding missing functionality to a web application, or creating mashups using content from multiple domains.

While Greasemonkey was originally a Firefox-specific extension, there are also ports of the extension to other browsers, like Tampermonkey for Google Chrome. According to the extension markets of Mozilla Firefox and Google Chrome, at the time of this writing, there are almost three million users who have the Greasemonkey and Tampermonkey extensions installed. Moreover, due to the popularity of the extension, a subset of the Greasemonkey functionality is, by default, supported in many modern browsers, where Greasemonkey scripts are treated as a special case of browser extensions.

In general, Greasemonkey scripts can be considered lightweight browser extensions. Users can write their own scripts, or find scripts written by other

```
// ==UserScript==
// @name         Hello World
// @description  Description of this script
// @namespace    http://author.com/gmscripts
// @include      http://example.com/*
// @include      http://*.example.com/*
// @exclude      http://login.example.com/*
// @require      http://author.com/lib.js
// @updateURL    http://author.com/hw.meta.js
// @downloadURL  https://author.com/hw.user.js
// @grant        GM_xmlhttpRequest
// ==/UserScript==

alert("Hello World");
GM_xmlhttpRequest({
  method: "GET",
  url: "http://www.shopping.com/",
  onload: function(response) {
    alert(response.responseText);
  }
});
```

Listing 4.1: Example of a Greasemonkey user script.

users, either dispersed on the web, or concentrated on community-driven script markets, much like the aforementioned popular extension stores. Greasemonkey scripts are different from other browser extensions, in that they target a different crowd of users. As further explained in the next section, Greasemonkey scripts are single-file JavaScript programs, without Manifest files and directory structures. Their lightweight nature allows them to be much more website-specific than normal browser extensions, e.g., disabling ads by hiding one specific HTML object on the user's favorite website, or game helping scripts for specific games on popular social networks. In addition, unlike browser extensions, the JavaScript nature of each script is not hidden in archive files. Instead, users can inspect and edit the code of their installed scripts from within the Greasemonkey extension.

4.2.2 Greasemonkey scripts

In this section, we demonstrate the basic structure and syntax of Greasemonkey user scripts, and the necessary concepts for the comprehension of the rest of the chapter.

Structure of scripts

Listing 4.1 shows a simple example of a user script. Notice that before the actual functionality of the script, there is script-specific meta-data in the form of comments enclosed by `// ==UserScript==` and `// ==/UserScript==`

The Greasemonkey engine will recognize the comments containing `@` signs and read-in the appropriate values. The `@name` and `@description` directives specify the title of a script and a user-readable description of what the script does. The `@namespace` directive allows for the separation of scripts that have the same filename. The `@include` and `@exclude` directives allow the script authors to specify the domains and webpages that their script should execute on. The `@require` directive allows the script author to include external JavaScript code and use it from the user script. Both `@updateURL` and `@downloadURL` are used during the automatic user script update process. The `@grant` directive specifies that the listed function should be added to the Greasemonkey sandbox.

The actual code of the user script starts where the meta-data comment block ends. In our example, the first call is to the standard `alert` function provided to JavaScript from the Browser Object Model and used to display message boxes to the user. The second function call, however, is towards a special Greasemonkey-specific function. *Greasemonkey API* functions have the `GM_` prefix and are typically able to do operations not allowed by standard JavaScript code. In this case, the script performs a cross-domain HTTP request to `http://www.shopping.com`, an operation that is otherwise forbidden by the *Same Origin Policy* (SOP), the browser's default security policy, for security and privacy reasons. Other Greasemonkey functions allow a user script to, among others, store and retrieve persistent data, access script-specific resources and register menu commands in the browser.

Figure 4.1 shows the Greasemonkey dialog that is displayed to the user trying to install our example user script. Notice that at the bottom of the dialog, the user is warned that the scripts can violate the user's security and privacy, and that the user is supposed to install scripts from only trusted sources.

Important meta-data

In this section, we expand upon some of the aforementioned Greasemonkey directives since these have security and privacy consequences.

`@include` As described earlier, Greasemonkey consults the `@include` directive to determine which pages a user script should be injected in. Greasemonkey

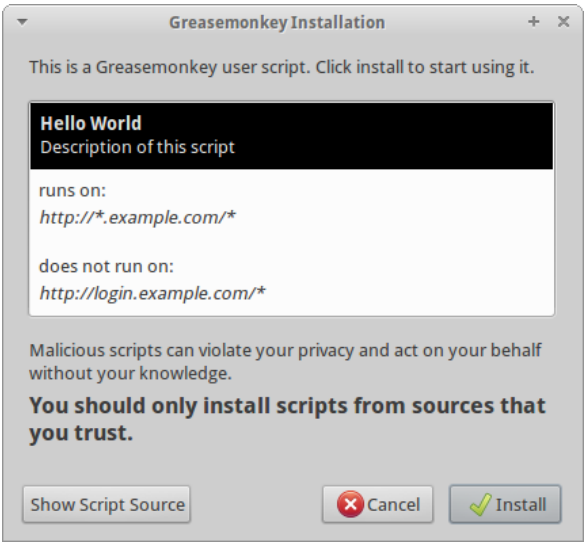


Figure 4.1: Greasemonkey Script Installation Dialog.

uses regular expressions to match the `@include` header against the entire URL, allowing a lot of flexibility. The script author might for instance add `@include http*://*.example.com` to allow the script to run on both HTTP and HTTPS versions of the example.com sub-domains, and the script author is even allowed to specify `@include *` to run the script on any website. If no `@include` directive is present, Greasemonkey will default to `@include *` for that user script.

`@require` Greasemonkey allows script authors to base their scripts on external JavaScript libraries through the `@require` directive in the script header. When a script with a `@require` directive is installed, the URL argument of this header is used to download the specified external JavaScript library and store it alongside the installed script. At runtime, the local copy of the external JavaScript library is executed together with the script code.

Consider the user script listed in Listing 4.1. During the installation of this script, Greasemonkey will find the `@require` header pointing to `http://author.com/lib.js`, download the referenced library script and store it alongside this user script for execution at runtime.

`@updateURL` and `@downloadURL` Greasemonkey has built-in functionality to automatically install updates for installed user scripts, when it detects that an update is available. To make use of this feature, script authors can specify the `@updateURL` and `@downloadURL` directives, as shown in Listing 4.1.

The `@updateURL` header is used to specify where the latest meta-data for a user script can be found. If this meta-data reveals the availability of a new version of a user script, the update process is triggered. The `@downloadURL` header lists the URL from which the updated script is to be downloaded, once the update process has been triggered.

If `@updateURL` or `@downloadURL` are not found in the script header, Greasemonkey automatically infers them from the location from which the script was installed. Both `@updateURL` and `@downloadURL` can use the HTTP scheme, but only when `@downloadURL` uses an HTTPS scheme, will the update be automatic.

`@grant` Based on the least-privilege principle, powerful functions like the ones in the Greasemonkey API should be available to user scripts, only if they are absolutely necessary. Recognizing the merits of this principle, Greasemonkey allows script authors to specify which functions of the Greasemonkey API should be added to the Greasemonkey sandbox, using the `@grant` header.

Consider again the user script listed in Listing 4.1, displaying the usage of this `@grant` header to request access to the `GM_xmlhttpRequest` function. The special directive `@grant none` is used to indicate that the script uses no Greasemonkey API functions at all, and thus none should be added to the sandbox. In the absence of `@grant` headers, Greasemonkey will attempt to infer the necessary API functions by analyzing the user script.

4.2.3 Attack surface

At this point, it should be evident that the extra functionality of user scripts, unfortunately comes with room for extra vulnerabilities. We consider three different attack scenarios: a) malicious user scripts abusing the pages in which they are injected, b) attackers abusing benign but vulnerable user scripts to attack webpages and, c) malicious pages trying to abuse the Greasemonkey engine and gain access to privileged functions.

In the first scenario, a victim installs a user script that advertises some functionality, e.g., automatically hiding ads on all webpages. This script may be a trojan horse which, next to hiding ads, steals private data from pages, the user's cookies, or even acts as a keylogger and captures all of the user's

keystrokes. Since JavaScript allows for extensive minification and obfuscation of code, identifying malice by simply inspecting the source code of a script can be an arduous and technically challenging procedure, which the majority of users will most likely not be able to perform.

In the second scenario, an attacker can take advantage of vulnerabilities introduced by user scripts on pages that otherwise would have no exploitable vulnerabilities, e.g., the exploitation of a DOM-based XSS vulnerability on a webmail application introduced by the added functionality of a Greasemonkey user script.

In the third scenario, an attacker can take advantage of user script vulnerabilities, not just to inject code in a benign page, but to inject code in Greasemonkey's sandbox. Greasemonkey makes use of sandboxing to protect the privileged `GM_` functions from possibly malicious scripts running on a website. Despite, however, of this sandbox and additional, stack-inspecting mechanisms of Greasemonkey, a poorly-written user script can still introduce unsafe code in the sandboxed environment, e.g., by `eval`-ing a string from a malicious page without performing the proper sanity checks. When this happens, a malicious script can, for instance, get access to the `GM_xmlhttpRequest` function of Greasemonkey which allows the attacker to send arbitrary requests towards any website, with the user's cookies embedded in them, and read the corresponding responses.

4.3 Community-driven script markets

An augmented browser extension, such as Greasemonkey, allows power users to create user scripts and use them in their daily browsing. Once written, a user script can be useful and generic enough, to be of value to other users. Script markets facilitate the sharing of such scripts by providing script authors with a disseminating platform and a feedback mechanism, and consumers of scripts with comments and ratings about the quality and utility of a particular script.

In this section, we discuss `userscripts.org`, the official script market for Greasemonkey scripts, and describe some general features and historical information. In addition, we also report on the building and categorizing of a dataset of user scripts and meta-data from `userscripts.org` that will be used throughout the rest of this chapter.

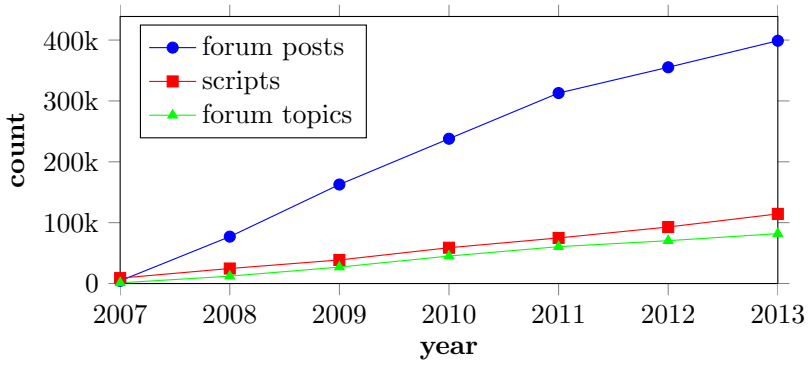


Figure 4.2: Historical data of `userscripts.org` on user scripts, forum topics and forum posts.

4.3.1 Userscripts.org

Userscripts.org is an online community established in 2005, which hosts community-provided Greasemonkey scripts and is the official script market associated with Greasemonkey.

The website allows members to upload, update and delete their user scripts. A forum hosted on the website allows the community members to communicate amongst themselves, discussing ideas and user scripts. User scripts can also be reviewed or flagged for further review by flagging them with issues. There are 5 categories of issues, namely "Broken", "Copy", "Harmful", "Spam" and "Vague." Scripts are characterized as "Vague" when the script authors do not adequately describe the purpose of their extensions. Members can vote on whether a flagged issue is present or not, and leave comments to support their vote.

The website also tracks several pieces of meta-data for each user script, among which, is a counter indicating how many times a user script was downloaded and installed. At the time of writing, the website hosts more than 114,000 Greasemonkey scripts written by more than 90,000 registered users. The websites forum contains about 400,000 forum posts spanning more than 82,000 forum topics.

Figure 4.2 plots data based on historical records [int] indicating the number of user scripts hosted by the community, the number of forum topics and forum posts since 2007. This data shows that the website has grown steadily since its creation. On average, the website has grown by about 48 user scripts, 37

forum topics and 179 forum posts per day, indicating that the Greasemonkey engine and its associated scriptmarket are active, despite the growth of more traditional browser extension markets.

4.3.2 Gathered dataset and statistics

To gain better insight into the user scripts provided by `userscripts.org`, we retrieved a total of 86,358 user scripts together with their accompanying meta-data.

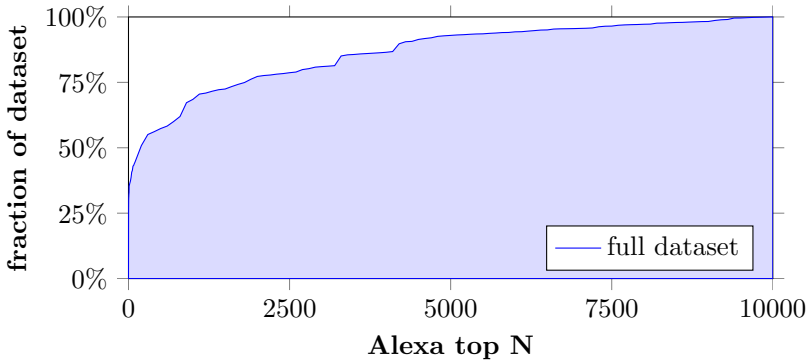


Figure 4.3: From the 37,893 user scripts for which the @include domain could be analyzed, 29.9% was designed for the Alexa top 3.

The authors of the user scripts in this dataset designed their scripts with a specific environment and purpose in mind. By analyzing the meta-data in each user script, we aimed to discover which websites these user scripts are meant to run on, as well as the category of each discovered website.

First, we correlated the meta-data of the 37,893 user scripts in our dataset that are designed for the Alexa [ale] top 10,000 "high-profile" websites.

Figure 4.3 shows the fraction of these "high-profile" user scripts related to the top Alexa domains they target. 11,313 (29.9%) of them are designed for the Alexa top 3 (Facebook, Google and YouTube), while 25,958 (68.5%) are designed for domains in the Alexa top 1,000.

Next, we used data from Trendmicro SiteSafety [tre] to split the user scripts into categories, according to the domain they are designed for. From the 37,893 "high-profile" user scripts in our dataset, 69.8% belong to the five categories shown in Table 4.1.

Table 4.1: Top five categories to which the "high-profile" user scripts belong, according to the domain for which they were designed.

Category	**count**	
Social Networking	12,238	32.3%
Search Engines / Portals	4,570	12.1%
Games	4,271	11.3%
Blogs / Web Communications	2,992	7.9%
Computers / Internet	2,396	6.3%
Total	26,467	69.8%

The data gathered from our dataset indicates that most user scripts are designed for "high-profile" websites aimed at entertaining and informing users, such as social networking sites, portals, games and blogs. These results show that many users are willing to install Greasemonkey scripts that operate on websites with valuable private data, like `facebook.com`. As we discuss in later sections, this willingness to trust user scripts can be abused by malicious script authors, in order to gain access to a user's private user data and perform actions on the user's behalf.

4.4 Malware assessment

Greasemonkey scripts are more powerful than traditional JavaScript programs, because they can manipulate and retrieve private data in a user's browser without SOP restrictions. Consequently, such scripts can be an attractive infection vector for malware authors, who can create malicious user scripts and trick users into installing them.

In this section, we discuss malware in Greasemonkey user scripts, why it is difficult to automate malware detection in user scripts, and how the `userscripts.org` community is currently attempting to deal with malware. We also analyze the subset of user scripts on `userscripts.org` that was labelled "harmful" by the community review process and provide some observations about this malware to improve the malware detection process.

```
1 var enemies = ["Cylon"];
2 for (e in enemies)
3     game.destroyColony(enemies[e]);
```

Listing 4.2: This Greasemonkey script can be both malicious or non-malicious, depending on whether the user is aware of its actions.

4.4.1 Defining and detecting malware in user scripts

Automatic malware detection is certainly a desirable mechanism that could, in theory, be used to protect Greasemonkey users from malicious user scripts, by screening new or updated user scripts, and marking them as malicious. Before this can happen, however, there first needs to be a clear definition of what exactly constitutes a malicious user script. A malicious script cannot just be defined as the presence of malicious code in the user script. The context and meta-data also need to be considered.

For instance, consider Listing 4.2, a code fragment inspired by an existing Greasemonkey script, containing a game helper, i.e., a script that assists a user while playing a specific game and gives him a competitive advantage over other users. When executed in a certain game, the code fragment destroys all colonies of type "Cylon."

If "Cylon" is an opposing team in the game, then the user would probably consider this script harmless and argue that the user script works as intended or advertised. If, on the other hand, the user's team name is "Cylon," then the script would sabotage her game and she would consider the script malicious. Without this additional contextual information, a malware detector would have to resort to the semantics of the script alone, and it would be unclear whether this code fragment should be classified as malicious or harmless.

To avoid such occasions, each script has a description field in its meta-data, where the script author can describe the purpose of the script and thus provide this additional contextual information. A malicious script could then be defined as a script whose semantics do not match its description and is performing some action that the user finds undesirable.

Unfortunately, verifying whether a user script's semantics match its description, is a task requiring non-trivial natural language processing, which in turn, relies on the user's verbosity and writing style. In addition, a user script's semantics

are context-specific, and requires a deeper understanding of the web-application it was designed for. As such, automating this malware detection process appears to be a difficult task, which is likely not to produce good results. Based also on our experience reverse-engineering Greasemonkey user scripts, we believe that the task of identifying malice should, at the moment, be left to human reviewers.

4.4.2 Userscripts.org issue reporting

The `userscripts.org` community website has a community-based, manual reviewing process to detect malicious user scripts. When a malicious user script is detected, the user can flag it as "harmful" and, optionally, explain her vote in the comment section.

In our dataset of 86,358 scripts, 626 (0.7%) are marked as "harmful" by at least one user of the `userscripts.org` community. Of those 626 scripts, 592 have at least as many votes in favor of "harmful" as votes against it. Due to the increased issue-related activity around these scripts, we focus on them for the rest of this section.

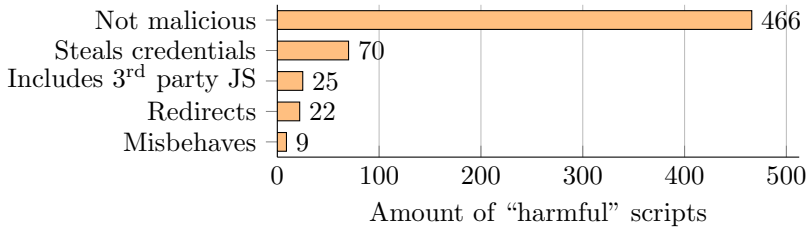


Figure 4.4: Categories of malware found in the 592 scripts labelled as "harmful" in the `userscripts.org` dataset. Almost 80% is harmless.

To determine the quality of this manual review process, we performed a manual analysis of these scripts to determine what users regard as "harmful." From the 592 "harmful" scripts, we could not find any trace of malice in 466 (78.7%) of them. For our purposes, we defined malice as the attempt to steal private data from a user, or trick the user into performing an action with potential monetary benefits for the attacker. We will refer to the remaining 126 scripts that do contain malware, as the *verified harmful dataset*.

A breakdown of the entire harmful dataset according to the reason the scripts were flagged, is shown in Figure 4.4. Ignoring, for the time being, the scripts

that we discovered to be not malicious, the largest fraction with 70 scripts contains malware designed to steal credentials in some form, from the user. This category contains scripts that steal cookies (29 scripts), steal username and password directly (22), steal username and password through phishing (14) and scripts that log and leak keystrokes (5).

The next largest fraction with 25 scripts include third party JavaScript into a loaded page. From the URL alone, it is not always clear what type of malware these scripts contain, if any. The included code could be dynamically generated and used to target specific users. We further investigate the possibility of targeted attacks in Section 4.5.

Twenty-two scripts simply redirect the user to another website, with the possible intent to lure the user into a drive-by-download scenario and install malware that way. Only five of these "redirect" scripts were reported by users to be the cause of a drive-by-download attack.

Finally, there remain nine scripts which simply "misbehave," and cannot be summarized in the previous categories. Their behavior is best described as making fraudulent transactions: sending spam on social networks, destroying online game assets, making a PayPal donation, ... etc.

Shifting our attention to the 466 benign scripts that were mislabeled as malicious, the main reasons for this labeling were "bad practices," e.g. providing custom update functionality instead of using the proper built-in functionality of Greasemonkey, and copied user scripts being mislabeled as "Harmful" instead of "Copy." For other scripts the issue reporter claims, among others, that the user script attracts copyright violations, destroys online communities, and even censors freedom of speech. These reasons indicate that the concept of "harmful" is not always clear to the members of the community, and that there should be a clearer definition.

In addition, we found some scripts that at one point included malware, but had the malware removed from the latest revision by its author. Such scripts, although now clean, still carry the "harmful" label because the labelling is not always reset on new revisions.

4.4.3 Malware observations

As mentioned earlier, a completely automatic malware-detection mechanism is not likely to produce good results for Greasemonkey scripts. However, from our manual review of user scripts in the verified harmful dataset, we observed certain patterns that kept on reoccurring in many of the malicious scripts.

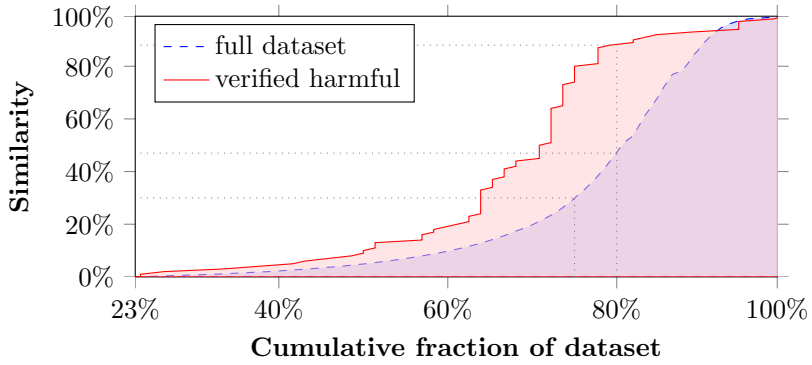


Figure 4.5: Cumulative amount of scripts in our datasets that are similar to older scripts of the full dataset. Notice that the verified harmful dataset contains more scripts with higher similarity to other scripts, than the full dataset.

From these patterns, we derived two detection methods that could assist human reviewers in prioritizing possibly malicious scripts, in the review process.

Malware insertion. During our analysis we observed that malware authors often copy an existing popular script and then add some malicious code, without modifying the original, surrounding code. The resubmitted malicious script is likely to appear during the search for scripts offering a specific functionality and installed by victim users, instead of the original script.

To determine the feasibility of detecting malware by identifying similarities between scripts, we set up an experiment to determine which scripts are copies of other scripts on `userscripts.org`. Comparing each script with every other script is a time-consuming process of complexity $O(n^2)$. Therefore, we limited the scope of our search to scripts of approximately the same size. For example, the size of the *Abstract Syntax Tree (AST)* of the largest piece of malware found during our manual analysis was 5,656 bytes. We doubled this amount and compared each script in our dataset to all older scripts with a maximum of 10KB size difference in the AST. This size-filter reduces the amount of comparisons by about 90%, from 3.7 billion to about 400 million.

Our comparison technique operates as follows: we consider that code is only inserted in one specific location in the script, and calculate what fraction of the new script is derived from an ancestor script.

Figure 4.5 shows the cumulative percentage of scripts in the full and verified

harmful datasets, ordered by their similarity with older scripts. For the full dataset (dashed line), we can see that almost 75% of the scripts have less than 30% in common with the other scripts.

Comparing this to the similarities of the verified harmful dataset (full line), we can clearly see that the malicious scripts are almost always "above" the full dataset, indicating a consistently higher similarity rating for most scripts in this dataset. For instance, we can see that 80% of the full dataset has a similarity rating of about 45%, while the verified harmful dataset has a similarity rating of about 90% for the same fraction of the dataset. Thus, the similarity of a new script with existing ones can be used to guide a human reviewer towards malicious scripts. Our findings are in line with the findings of Kapravelos et al. [HNL14], who notice that authors of traditional JavaScript malware try to evade detection by copying popular JavaScript libraries (like jQuery) and injecting them with malicious code.

Malware reuse. In addition to copying popular user scripts, we also observed that malware authors occasionally recycle malware fragments. Motivated by this observation, we ran the following experiment to uncover additional text-strings that are indicative of malware. From the set of all verified harmful scripts, we extracted all strings of length ten or more, comprised of alphanumeric characters plus '.', '-' and '_'. We then searched for these strings in the full dataset.

Table 4.2: Five text-strings appearing in ten or more scripts, of which at least 50%, but less than 100% are verified harmful.

Text-string	**total**	**harmful**	
`voxDve.indexOf`	7	6	(85.7%)
`voxDve.substr`	7	6	(85.7%)
`xVDs.iterateNext`	7	6	(85.7%)
`eleNew.nextSibling`	23	20	(87.0%)
`eleNew.parentNode.insertBefore`	23	20	(87.0%)

From the results of this experiment, we only retained those strings which occur in ten or more scripts, of which at least half are verified harmful, yielding a total of 18 strings. For brevity, we only show five of those in Table 4.2.

The string `eleNew.parentNode.insertBefore` was found in 20 scripts in the "harmful" dataset all of which were associated with a malicious cookie-grabber. There are, however, 23 scripts in the full dataset that contain this specific string. The extra three also contain the malware but were not flagged by the community.

This experiment indicates the value of a simple text-search for the community's review process. As we have shown, it is straightforward to extract indicative strings from a base set with known malware. Since a text-search on new scripts for these text-strings is equally simple, this detection method is very effective.

4.5 Targeted attacks

In Section 4.4 we considered that a malicious script author can add malware to a user script. This method has some drawbacks from the malicious author's perspective. The main one is that the malware-containing script has to be uploaded to the script market and is thus available for analysis by the rest of the community.

Another method of spreading malware, which does not require exposing the malware to an online community, is by infecting users during installation of the script, or during the update process. A malicious script author, using one of these methods, can effectively infect users with malware without exposing the malware to an online community and even allow him to cherrypick which users to infect, leading to targeted attacks.

In this section, we discuss these targeted attacks during installation and during the update process In addition, we measure how many scripts in our dataset are susceptible to these attacks.

During user script installation. Consider that the author of the example listed in Listing 4.1 is malicious and is determined to target a new user of his user script with malware by taking advantage of the `@require` directive.

A review of this user script by the community could show that the script, by itself, does nothing harmful. To review the `@required` JavaScript library, the reviewer would need to download the library from `author.com`, whose server-side code could determine that it is under review and return harmless code. The review of this downloaded JavaScript library will then equally indicate it is harmless.

Reassured by the community review, the targeted user could decide to install the script. During installation of this script, a request will be sent from the user's browser towards `author.com`, requesting the specified JavaScript library. At this point, code running on that webserver can again determine where the request is coming from, e.g. by geo-locating the IP address or fingerprinting the user's browser [Eck10], and reply with custom malware for the targeted user.

Through user script updates. Similarly, the Greasemonkey update process can be abused by a malicious script author to infect a targeted user of his user script, with malware.

Consider again the example in Listing 4.1 where both `@updateURL` and `@downloadURL` headers are used. At regular intervals, Greasemonkey will initiate the update process for all installed user scripts. If the update request originates from a targeted user, the server-side can pretend there is an update available and push malicious code to the Greasemonkey extension, a fact which will be invisible for any other user of that user script.

Appearance in the dataset. Although these attacks are possible, we cannot easily detect whether our dataset contains user scripts that covertly install malware during the installation or update process. Such scripts, after all, would be specially crafted to resist this kind of review. Nevertheless, we are interested in discovering user scripts in our dataset which could be used to covertly install malware.

In the full dataset, 10,866 (12.6%) scripts have a valid `@require` directive. Of these, 3,264 scripts `@require` JavaScript exclusively from `userscripts.org`, 6,897 download them exclusively from third-party domains, and 705 use both. This means that 7,602 scripts (8.8% of the full dataset) `@require` JavaScript libraries from third-party domains and may covertly install malware during the installation process.

The three most popular third-party domains from which external JavaScript is loaded are `googleapis.com` (3,738 scripts), `sizzlemctwizzle.com` (1,339 scripts) and `googlecode.com` (868 scripts). The most popular user script in our dataset, which `@require`s an external JavaScript library is a Farmville script with over 60 million installations, and `@requires` JavaScript from `sizzlemctwizzle.com`.

Although these domains can be considered trusted due to their popularity, there are many third-party domains that only occur a handful of times in an `@require` directive in the dataset, indicating that they are most likely tied exclusively to the script's author. Such domains can potentially serve malware covertly.

Shifting our focus to the update mechanism, 1,135 user scripts provide a valid `@updateURL`, 516 provide a valid `@downloadURL` and 481 provide both. As mentioned in Section 4.2.2, the remaining 85,188 scripts without either a `@updateURL` or `@downloadURL` have the respective URL derived from the location from which the script was installed, which in this case is `userscripts.org`.

From the 516 scripts that provide a valid `@downloadURL`, 462 are located

in the `userscripts.org` domain, while 54 point elsewhere. Most of the `@downloadURL`s use the HTTPS scheme (371) while 145 use HTTP. This data shows that from the 516 scripts with an explicit `@downloadURL` in our dataset, 334 point to `https://userscripts.org` and will automatically update whenever an update is available. For all the rest, the Greasemonkey engine will automatically set the `userscripts.org` domain as the update domain accessible using the HTTPS scheme, meaning that for the vast majority of scripts, updates will be performed silently.

The three most popular third-party domains from which updates are downloaded, both for HTTP and HTTPS, are `github.com` (27 scripts), `zanloy.com` (4 scripts) and `google.com` (3 scripts). The most popular script in our dataset, which updates from a third-party domain over the HTTPS protocol, is an IMDB script with more than 50,000 installations, updating from `https://github.com`.

4.6 Attacking weak scripts

In the previous section, we discussed scripts that are malicious by design, giving their authors the ability to harm those scripts' users. Because Greasemonkey injects user scripts into visited webpages, these user scripts unfortunately increase the attack surface of the user. Thus, even if scripts are not malicious by commission, they may still cause harm to their users due to vulnerabilities, by omission.

In this section, we discuss two vulnerabilities that occur in user scripts: DOM-based XSS and overly generic `@include` directives. Through these vulnerabilities, an attacker can trick a victim's browser into executing code on webpages onto which a user script acts, or even any webpage he wants, and potentially even gain access to powerful Greasemonkey API functions.

4.6.1 DOM-Based XSS

DOM-Based XSS, is an XSS attack in which a payload is executed that is somehow stored in the DOM of the victim's browser. This is in contrast with reflected or persistent XSS, where the payload is placed inside the visited website.

Consider the example shown in Listing 4.3, which appends a newly created `div` tag to the loaded webpage and writes the current page's location into it. This code fragment contains a DOM-based XSS vulnerability because it allows an

```
var d = document.createElement("div");
d.innerHTML = "This page is located at " + document.location.href;
document.body.appendChild(d);
```

Listing 4.3: Example script vulnerable to DOM-based XSS.

```
<div>
This page is located at http://example.com/?<script>alert(1);</script>
</div>
```

Listing 4.4: Resulting div of DOM-based XSS attack.

attacker-controlled string to be inserted into the HTML page of the currently loaded website.

If this code fragment is used on `http://example.com/`, a victim's browser visiting `http://example.com/?<script> alert(1);</script>` would generate the HTML code shown in Listing 4.4. The attacker payload, in this case `alert(1);` would be executed as JavaScript in the `example.com` origin.

The DOM-based XSS vulnerability in the previous example is restricted to the webpage on which the code in Listing 4.3 is present. Using the same code in a Greasemonkey script potentially lifts this restriction. The vulnerability will then be injected into any page on which the Greasemonkey code is included. An attacker with knowledge of this situation, has thus a much larger target: every page on which this Greasemonkey script is executed, becomes vulnerable.

DOM-based XSS analysis setup. To determine whether any DOM-based XSS vulnerabilities occur in our user scripts dataset, we screen all scripts using a lightweight static-analysis method. Using the Parser API [par] in SpiderMonkey [spib], Mozilla's standalone JavaScript engine, we parsed all scripts in our dataset and obtained a simplified AST for each one of them. Using the list of sources and sinks listed in Figure 4.6, we searched for sources used directly in the argument list of sinks. As such, all the results reported in the next sections are lower bounds of vulnerabilities.

Table 4.3: Breakdown of amount of scripts with detected DOM-based XSS vulnerabilities according to the used sources and sinks. Only those sources and sinks with any results are shown. (*) the totals do not reflect the sum on each row, but rather the amount of total unique scripts for the given sink.

	`document`								`window.name`	**Total**
	`.body`	`.cookie`	`.forms`	`.links`	`.location`	`.title`	`.URL`	*other*		
`document.write(x)`	0	0	0	0	1	1	0	1	0	3
`eval(x)`	3	2	1	0	0	0	0	73	0	79
`e.innerHTML = x`	721	4	5	17	83	14	6	800	5	1,654 (*)
Total	724	6	6	17	84	15	6	874	5	1,736 (*)

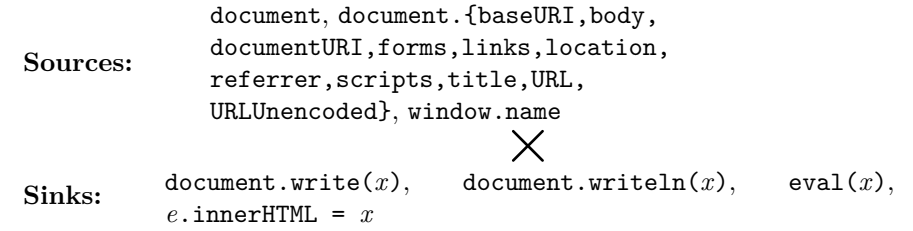


Figure 4.6: Sources and sinks used in the lightweight static analysis performed to look for DOM-based XSS.

Results. The results of our DOM-based XSS analysis on the full dataset retrieved from `userscripts.org`, are shown in Table 4.3. From the 86,358 scripts in our dataset, our analysis revealed 1,736 that contain a DOM-based XSS. The majority of scripts are vulnerable through the `e.innerHTML` sink (1,654 or 95.3%) and the various sources originating from the `document` object (99.7%).

Note that not all sources are under the control of any attacker and might require the ability to place persistent data onto a website. The four sources that can be influenced by an attacker are `document.cookie`, `document.location`, `document.URL` and `window.name`. From the dataset, 101 scripts are vulnerable to DOM-based XSS involving those four sources.

The most prominent, vulnerable to DOM-based XSS, user script that we discovered is the fourth most popular script on the `userscripts.org` script market, with almost 40 million installations. The script is designed for a popular massively multiplayer online strategy game called *Ikariam*. We created a proof-of-concept exploit where, through the clicking on a specially-crafted URL, similar to the one used in the example in the previous section, we could inject JavaScript in authenticated pages of users.

4.6.2 Overly generic `@include`

As explained in Section 4.2.2, the `@include` directive specifies which webpages a user script is injected in. The `@include` directive allows the use of a wildcard, and uses regular expression matching to test the entire URL of the webpage being visited.

If the `@include` wildcard is used in a too generic way, this can lead to a security problem. For instance, reconsider the introductory example in Listing 4.1.

In this script, the directive `@include http://*.example.com/*` is used. An attacker might craft the URL `http://www.mybank.com/# x.example.com/abc` and trick a user of this script to visit it. Greasemonkey's regular expression will then match the `@include` directive against this crafted URL and falsely assume that the author of the script wants the script to be executed on `http://www.mybank.com/`. The attacker has caused the script to run on a webpage for which it was not intended, by abusing the `@include` wildcard.

`@match` The developers of Google Chrome, in their adaptation of the Greasemonkey engine, recognized that the wildcard `*` in the `@include` directive, was not strict enough and could lead to insecure situations. For this reason, they created the `@match` [chr] directive which provides the same functionality as `@include`, but in a safer way.

Google Chrome's `@match` wildcard is context-sensitive and is applied by splitting a URL into three parts: a scheme, a host and a path. A `*` wildcard can occur within each part, but cannot match anything that violates the borders between the parts. For instance, in the directive `@match http://*.example.com/about.html` the wildcard is located in the host part and can only match characters associated with a host. Unlike with `@include`, the `http://www.mybank .com/#x.example.com/about.html` URL will not be matched, since `/` and `#` are not valid characters for a hostname. Likewise, the wildcard in `*://www.example.com/` can only match `http` or `https`.

To be compatible with user scripts for Google Chrome, Greasemonkey adopted the `@match` directive alongside its `@include` directive. In cases where both `@include` and `@match` directives are used, the `@include` directive is handled first.

Table 4.4: `@include` and `@match` directive usage, "insecurely" means an overly generic `@include`.

	`@match`	No `@match`	Total
`@include` securely	670	33,775	34,445
`@include` insecurely	770	39,955	40,725
No `@include`	884	10,304	11,188
Total	2,324	84,034	86,358

Usage of the `@include` and `@match` directives Table 4.4 divides the scripts in our dataset with regard to `@include` and `@match` directives. From the 86,358 scripts in our dataset, 75,170 (87.0%) contain a `@include` directive, of which

40,725 insecurely with a too generic wildcard. Since scripts without an explicit `@include` directive automatically obtain a `@include *` directive, this means that 51,913 scripts or 60.1% of the full dataset can be tricked into executing on a different domain than the one they were designed for.

Only 2,324 specify a `@match` directive, of which 1,440 also specify an `@include` directive. Of those 1,440, 770 have insecure `@include` directives, meaning the `@match` directive's security advantage over a `@include`, is completely negated.

The most popular script with an unsafe `@include` directive is, at the same time, the most popular script on `userscripts.org`, a social networking script with more than 250 million installations. It uses an overly generic wildcard `@include` directive of the form `@include http://*website.com/*`.

4.6.3 Resulting malicious capabilities

Global XSS The combination of a DOM-based XSS vulnerability, and an overly generic `@include` directive, results in a critical vulnerability. Scripts which contain this combination of vulnerabilities allow an attacker to execute malicious code on any webpage that the attacker chooses, by crafting a specific URL.

From the 1,736 vulnerable scripts revealed from our analysis to be vulnerable to DOM-based XSS vulnerabilities, 944 (54.3%) also use overly generic `@include` directives and can thus be used to perform global XSS attacks.

Privileged XSS The case of a DOM-based XSS where attacker-controlled data find its way into an `eval(x)` sink reveals an extra security issue because it allows malicious code to execute inside the Greasemonkey sandbox. As explained in Section 4.2.3, malicious websites can leverage such a DOM-based XSS vulnerability in a user script, to gain access to the Greasemonkey API.

Consider for instance the example in Listing 4.1. The example script uses `GM_xmlhttpRequest` to get access to cross-origin resources from `http://www.shopping.com/`. This API function will be present in the sandbox where the user script executes, because `@grant GM_xmlhttpRequest` is used to request it. If this example script also contained a DOM-based XSS vulnerability with an `eval(x)` sink, then a malicious website could trigger this vulnerability, executing code inside the Greasemonkey sandbox and get access to the powerful `GM_xmlhttpRequest` function.

From the 79 scripts that contain a DOM-based XSS with an `eval(x)` sink, 60 execute in a sandboxed environment with access to the Greasemonkey API and can thus leak that API to a malicious website which may abuse it.

Privileged, global XSS To aggravate the problem further, it is possible to combine all three vulnerabilities: a script with an overly generic `@include` directive, vulnerable to a DOM-based XSS attack where attacker-controlled data flow into `eval(x)`, thereby exposing the Greasemonkey API.

A script such as this can be abused by an attacker by luring victims to a specially crafted URL, causing attacker-controlled code to be executed, with access to the powerful Greasemonkey API. Since the Greasemonkey API functions are not bound by the Same Origin Policy, an attacker could then abuse them to steal private data from the victim's browser, across all sites. From the 60 scripts we identified as being vulnerable to a DOM-based XSS with an `eval(x)` sink and which also expose the Greasemonkey API, 58 use an overly generic `@include` directive.

The most prominent example is a script installed by 1.2 million users, which, even though is meant to run on a gaming site, can be forced to run on any website, due to its overly generic use of wildcards. Moreover, the script makes insecure use of `eval` allowing an attacker to execute arbitrary code in the Greasemonkey sandbox. We created a proof-of-concept exploit which amounts to a Man-in-the-Browser attacker, i.e., we can conduct requests towards all websites (together with the user's cookies), read the responses, and inject malicious JavaScript on any domain.

4.7 Related work

To the best of our knowledge, this chapter is the first one that tries to shed light on alternative, community-driven, JavaScript markets. Closely related, however, is research done in identifying malicious and vulnerable browser extensions from the official extension markets of Mozilla Firefox and Google Chrome.

Barth et al. [BFSB10] criticize the all-permissive Firefox extension system showing that only three out of 25 investigated extensions required full system access. The authors propose an alternative extension architecture that requires extensions to explicitly ask permission for access to resources and also compartmentalize the browser so that a vulnerability in a "benign-but-buggy" extension does not necessarily mean arbitrary code execution with the permissions of the user running the browser process. Guha et al. [GFLS11] study

the Google Chrome market and show that a significant fraction of extensions request more permissions than they require. The authors set out to create a more fine-grained policy system to describe access to resources, as well as a statically-verifiable, platform-independent language for writing extensions which are then automatically compiled to JavaScript and other platform-dependent code.

Liu et al. [LZYC12] remind that next to benign-but-buggy extensions, malicious extensions pose real threats to the security and privacy of users. The authors present some proof-of-concept extensions that send spam emails, steal bank credentials, and perform distributed denial-of-service attacks on demand. As a defense against malicious extensions, the authors propose the use of micro-privileges, such as `inject_script` and `cross_site` in order to further increase the granularity of Chrome's fine-grained policy system.

VEX [BKMW10] analyzes Firefox extensions, such as Greasemonkey, for privilege escalation vulnerabilities, but does not analyze the user scripts used by Greasemonkey itself.

While malicious browser extensions are typically written in JavaScript, malicious JavaScript, today, has a different connotation, that of code which exploits some vulnerability in the browser or in one of the browser plugins to eventually lead to drive-by downloads, i.e., achieve remote code execution and download arbitrary malicious executables on the victim's machine. According to a recent study by Barracuda Labs, the visitors of the 25,000 most popular sites on the Internet, got exposed to more than 10 million such exploits, on February of 2012 alone [Bar12]. Due to the great magnitude of the problem, there has been a significant body of research in detecting malicious JavaScript, using honeypots [WBJ+06], dynamic analysis of JavaScript code [CKV10, RLZ09, KLZS12, HNL14], and hybrid systems [CLZS11, RKD10] which utilize both static and dynamic techniques to analyze JavaScript code. Purely static analysis of JavaScript has met with limited success due to the large degree of obfuscation that can be achieved in the JavaScript language. As such, purely static techniques [CCVK11] are best used as lightweight filters which can separate the "definitely benign" from the "possibly malicious." The latter can be used as input in more resource-intensive dynamic systems while the former can be safely ignored.

The main difference of this type of malicious JavaScript with the types of malicious Greasemonkey scripts analyzed in this chapter, is that in our case, maliciousness is context-specific. Thus malicious Greasemonkey scripts are more likely to interact in an abusing way with a specific web application, rather than trying to trigger a vulnerability in the browser. As such, they may be only discoverable when the user is on a specific page of a specific website, making dynamic detection of context-specific maliciousness significantly harder to define

as well as detect.

Typical JavaScript sandboxing techniques [MSL+08, LGV10, ARD+11, AVAB+12, IW12, MPS10] attempt to isolate malicious code in a controlled environment and prevent references to powerful functionality from leaking inside the sandbox. In contrast, Greasemonkey creates a sandbox with its powerful API inside and attempts to prevent the leakage of references to this API to the outside. The vulnerabilities exposed in this chapter allow an attacker in some cases to inject malicious code inside the sandbox, causing a situation similar to the "inverse sandbox" effect described in [ARD+11].

4.8 Conclusion

As more and more applications move from the desktop to the web, power users turn to augmented-browsing tools, to personalize their web applications.

In this chapter, we analyzed the Greasemonkey browser extension and the `userscripts.org` script market, searching for evidence of malware and vulnerabilities, as well as documenting the ways with which community-driven script markets deal with malicious scripts. Through this process, we find that automated malware detection in a script market is difficult because of the context-sensitive nature of malice, and that the review process of `userscripts.org` is ineffective in 78% of the cases. Next to the discovery of malicious scripts, we identify ways in which malicious authors can bypass the community review process and covertly infect user script users with malware.

Moreover, we identify and analyze two types of vulnerabilities found in user scripts, which could allow an attacker to use the restricted and powerful Greasemonkey functions to, among others, bypass the Same Origin Policy, and force a user script to run on any website.

We found that DOM-based XSS vulnerabilities are present in 2% of user scripts and that 60.1% of user scripts can be forced to run on any webpage. Finally, we show how an attacker can combine many vulnerabilities to launch powerful privileged, global XSS attacks and discover 58 scripts that are susceptible to this attack. We demonstrate this attack through a proof-of-concept exploit for one of these user scripts, installed by over a million users, allowing us to steal their data across all sites.

The purpose of our work is to highlight the inherent difficulties of securing script markets against malicious actors, and the possibility of weaponizing benign scripts against otherwise secure websites.

Responsible disclosure

We are in the process of disclosing these vulnerabilities to all involved parties.

Acknowledgements

This research was performed with the financial support from the Prevention of and Fight against Crime Programme of the European Union (B-CCENTRE), the Research Fund KU Leuven, the FP7 projects STREWS, NESSoS and WebSand, and the IWT project SPION.

Chapter 5

JavaScript Sandboxing

"Are you quite sure that all those bells and whistles, all those wonderful facilities of your so called powerful programming languages, belong to the solution set rather than the problem set?"

— Edsger W. Dijkstra, renowned computer scientist

"You cannot trust code that you did not totally create yourself."

— Ken Thompson, creator of the UNIX OS

The gadget attacker, as defined in Section 2.3.3, has the ability to integrate a malicious gadget into a trusted web application. This allows the attacker to execute any chosen JavaScript code in the JavaScript execution environment of this trusted web application's origin and access its sensitive resources.

Given this attacker model, we cannot stop the attacker from presenting a web application user's browser with chosen JavaScript. Instead, we can isolate the JavaScript and restrict its access to certain resources and functionality, by executing this code in a JavaScript sandbox.

From the typical web scenario architecture from Section 2.3.3 shown again in Figure 5.1, keeping in mind our attacker model, there are only two possible locations that can be considered to deploy a JavaScript sandboxing mechanism: the trusted web application and the client's browser. The third-party script provider is considered untrustworthy.

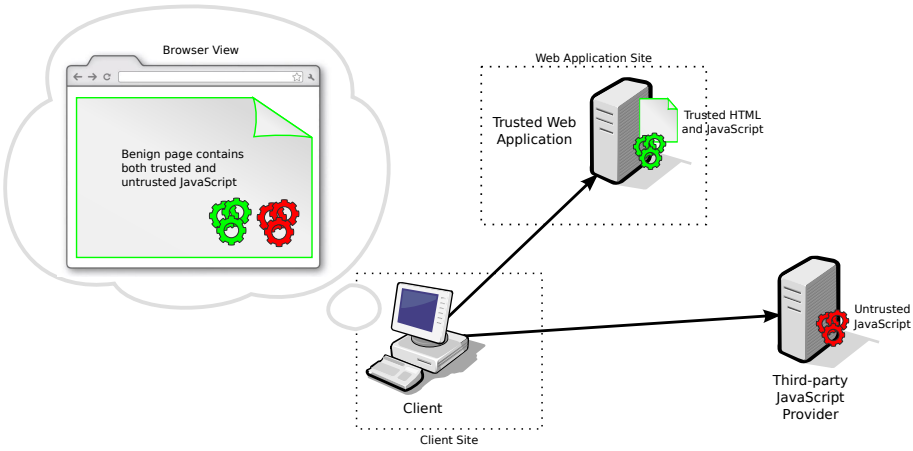


Figure 5.1: A typical web application with third-party JavaScript inclusion. The web application running in the browser combines HTML and JavaScript from a trusted source, with JavaScript from an untrusted source.

The developer of the web application and the server hosting it, are trusted according to the attacker model. This server then offers a possible location to facilitate JavaScript sandboxing. Before serving the untrusted JavaScript from the third-party script provider to the client, the code can be reviewed and optionally rewritten to make sure it does not abuse the web application's available resources.

The client's browser provides a second location to sandbox JavaScript, because it is also considered trusted. With direct access to the JavaScript execution context, a JavaScript sandboxing system located at the client-side has better means to restrict access to resources and functionality.

The remainder of this chapter is organized as follows. Section 5.1 discusses JavaScript sandboxing systems involving JavaScript subsets and rewriting systems. Section 5.2 discusses browser modifications to achieve JavaScript sandboxing. Section 5.3 discusses JavaScript sandboxing systems which do not require any browser modifications. Section 5.4 concludes this chapter with a brief discussion of the advantages and disadvantages of the three categories of JavaScript sandboxing systems.

```
var cmd = prompt();

// MD5 algorithm computes a one-way hash
function md5(m) {
    // ...
    return m;
}

// verifies whether the given input is "alert('hello')"
if(md5(cmd) == "3b022ec21226e862450f2155ef836827") {
    eval(cmd);
}
```

Listing 5.1: Example JavaScript calling eval() on user input, but only if its MD5 hash matches a given hash.

5.1 JavaScript subsets and rewriting

JavaScript is a very flexible and expressive programming language which gives web-developers a powerful tool to build web-applications. However, this same powerful tool is also available to attackers wishing to execute malicious JavaScript code in a website visitor's browser.

Moreover, the powerful nature of JavaScript is problematic because it hinders code verification efforts which could prove safety properties for a given piece of JavaScript code.

Example: eval()

Consider for instance the JavaScript fragment in Listing 5.1. When executed in a browser, this code will prompt a user to input a line of text. The one-way hashing algorithm MD5 is then used to compute a hash of this line of text. If the hash matches "3b022ec21226e862450f2155ef836827", the MD5 hash for "alert('hello')", then the line of text is passed to the eval() function and executed as JavaScript code.

Given that the MD5 hashing algorithm cannot easily be reversed, it is practically impossible for a code verification tool to automatically determine the effect of this code, prior to its execution. The eval() function illustrates a feature of JavaScript which makes code verification difficult because of its dynamic nature. For this reason, eval() is considered evil [RHBV11] and should be used with the greatest care, or not be used at all.

```
var o = {f:2, x:4};

console.log("before with: f == " + f);
console.log("before with: x == " + x);
console.log("before with: \"x\" in window == " + ("x" in window));

with(o) {
    function f() { }
    console.log("inside with: f == " + f);
    var x = 3;
    console.log("inside with: x == " + x);
}

console.log("after with: o.f == " + o.f);
console.log("after with: o.x == " + o.x);
console.log("after with: f == " + f);
console.log("after with: x == " + x);
console.log("after with: \"x\" in window == " + ("x" in window));
```

Listing 5.2: Example JavaScript using the "with" construct to place a new object at the front of the scope chain during the evaluation of the construct's body. This example is adapted from Miller et al. [MSL+08].

Example: Strange semantics and scoping rules

As another example, the JavaScript fragment in Listing 5.2 illustrates some strange semantic rules in JavaScript, including the "with" construct. This particular example showcases some non-intuitive scoping rules associated with the scope chain. The scope chain consists of an ordered list of JavaScript objects which are consulted when unqualified names are looked up at runtime.

Before continuing, the reader is advised to read the code and try to predict what it will output. The actual output of the code in this example, is listed in Listing 5.3.

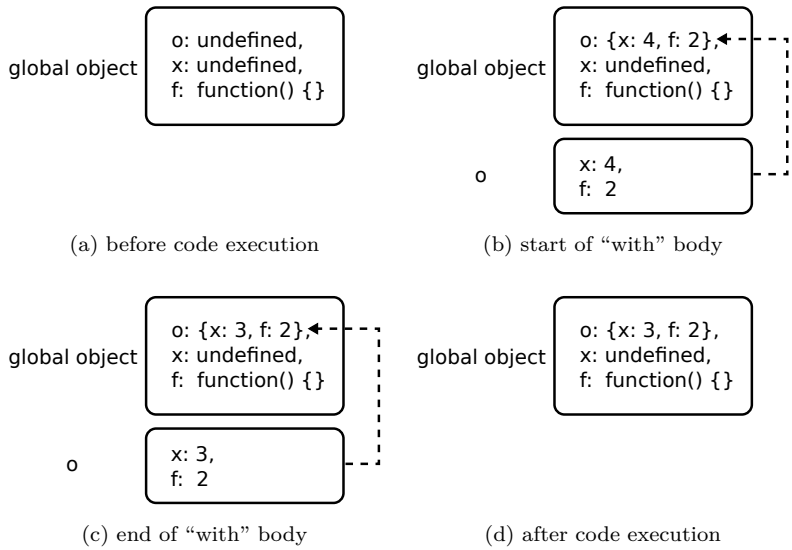
From the output, it appears that both "f" and "x" are already defined before they are even declared, but "x" has "undefined" as value. Using "with", the user-defined object "o" is pushed to the front of the scope chain. The new function "f()" is declared, but the subsequent "console.log()" call seemingly is not aware it. Instead, the value of "f" is retrieved from the first object in the scope chain ("o"), resulting in "2." Then, a variable "var x" is declared and assigned "3." The following "console.log()" call is aware of this declaration and outputs the correct value. Outside the with loop, the object "o" has changed to reflect the new value of "o.x", but did not record any change to "o.f".

The strange behavior in this example indicates that variable and function

```
before with: f == function f() { }
before with: x == undefined
before with: "x" in window == true
inside with: f == 2
inside with: x == 3
after with: o.f == 2
after with: o.x == 3
after with: f == function f() { }
after with: x == undefined
after with: "x" in window == true
```

Listing 5.3: Output of example in Listing 5.2.

declarations have different semantics in JavaScript. The discrepancy between variable and function declarations can be explained by a process called "variable hoisting." Variable hoisting examines the JavaScript code to be executed and performs all declarations before any code is actually run.



(a) before code execution

(b) start of "with" body

(c) end of "with" body

(d) after code execution

Figure 5.2: The scope chain during execution of the example in Listing 5.2. In this depiction, the scope chain grows down so that newly pushed objects are at the bottom.

A graphical representation of the scope chain during the execution of this

example is shown in Figure 5.2 and can be used as a visual aid during the explanation.

Depicted in Figure 5.2a is the result of the variable hoisting before any code is run. The function "f()" and the variable "x" are declared on the global object. While the variable "x" has value "undefined", the function "f()" is declared and is assigned its value immediately.

Next, the object "o" is pushed to the front of the scope chain. The scope chain right after this push and right before the start of the "with" construct, is shown in Figure 5.2b. Any unqualified names are now looked up in the variable "o".

The third image shown in Figure 5.2c, depicts the state of the scope chain at the end of the "with" body. Here, the value of the property "x" of the object "o" has changed to "3" because of the assignment. Also note that the value of "f" has not changed because variable hoisting declares and initializes a function in a single step before the code is run, and so outside of the "with" body.

Finally, in Figure 5.2d, the scope chain is restored because the "with" body ended.

The strange scoping rules and semantics of "with" are difficult to reason about for uninitiated programmers. Widely-acknowledged as being a "JavaScript wart" [GSK10], it is often recommended to not use the "with" construct because it may lead to confusing bugs and compatibility issues [Mozg].

JavaScript subsets: verification and rewriting

The goal of JavaScript code verification and rewriting is to inspect JavaScript code before it is executed in a browser, and ensure that it is not harmful.

In the light of the previous examples, it can be desirable to eliminate those constructs from the JavaScript language that hinder code verification efforts or cause confusion in general. At the same time, it is also desirable to maintain as much of the language as possible so that JavaScript is still useful. Such a reduced version of JavaScript, with e.g. "eval()" and "with" construct missing, is called a JavaScript subset.

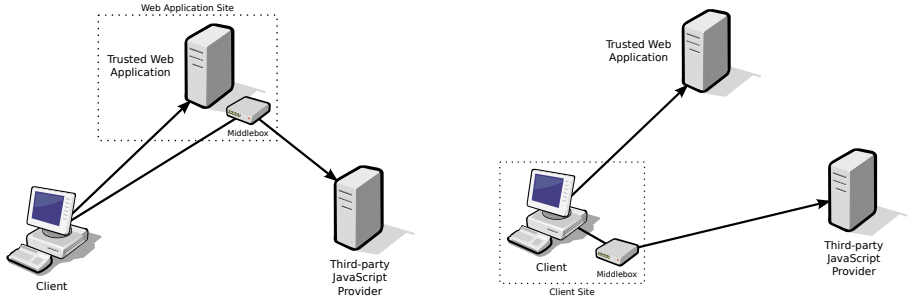
The usage of a JavaScript subset must be accompanied by a mechanism which verifies that a given piece of code adheres to the subset. A deviation from the subset's specification can be handled in two ways: rejection and rewriting.

Rejection is the simpler of both options, treating a deviation from the subset as a hard error and refusing to execute the given piece of code.

Rewriting is a softer alternative, transforming the deviating piece of code into code which conforms to the subset. Such a rewriting phase can also introduce extra instrumentation in the code to ensure that the code behaves in a safe way at runtime.

Interception in a middlebox

Both the JavaScript subset verification and rewriting steps necessitate the processing of raw third-party JavaScript code before it reaches the client's browser. These steps are to be performed in a *middlebox*, a network device that sits on the network path between a client and a server. Such a middlebox may consist of a physical device unrelated to either client or server, but it may just as well be collocated with either client or server.



(a) Middlebox at web application site

(b) Middlebox at client site

Figure 5.3: Architectural overview of a setup where a middlebox is used for code verification and transformation, at the web application site and at the client site.

From the attacker model discussed in Section 2.3.3, we can eliminate the third-party script provider's site as a possible location to verify and rewrite JavaScript. We are left with two possible locations for these tasks: the site of the trusted web application and the client's site.

A middlebox at the site of the web application, as shown in Figure 5.3a, can equally be implemented as part of a separate network device such as a load-balancer, reverse proxy or firewall, or can be integrated to be part of the web-application.

A middlebox at the client's site, as shown in Figure 5.3b, can either be a implemented as a proxy performing the required verification and translation

```
"use strict";

var example = 123;
// the following fails because the name is misspelled
exmaple = 345;

// the following fails because of a duplicate key name
var obj = {p:1, p:2};

// the following fails because "with" is not allowed
with(obj) {
    alert(p);
}
```

Listing 5.4: JavaScript strict mode example.

steps, or as a browser plugin or extension, implementing the proxy's behavior as part of the browser.

ECMAScript 5 strict mode

ECMAScript 5 strict mode [Mozb], or JavaScript strict, is a standardized subset of JavaScript with intentionally different semantics than normal JavaScript.

To use strict mode, a JavaScript developer must only place "use strict"; at the top of a script of function body, as shown in Listing 5.4. Strict mode will then be enforced for that entire script, or only in the scope of that function. JavaScript strict mode can be mixed with and function together with normal JavaScript.

Strict mode removes silent failures and turns them into hard errors that throw exceptions and halt JavaScript execution. For instance, accidentally creating a global variable by mistyping a variable name, will throw an error. Likewise, overwriting a non-writable global variable like "NaN" or defining an object with a duplicate key, causes strict mode to throw errors.

Strict mode simplifies variable names and allows better JavaScript engine optimization by removing the "with" construct. Through this construct, JavaScript engine optimizations may be confused about the actual memory location of a variable. In addition, strict mode changes the semantics of "eval()" so that it can no longer create variable in the surrounding scope.

Strict mode also introduces some fixes with regard to security. It is no longer possible to access the global object through the "this" keyword, preventing

```
1 // original JavaScript code
2 eval("...");

3 // rewritten by BrowserShield
4 bshield.invokeFunc(eval, "...");
```

Listing 5.5: Example JavaScript code rewritten by BrowserShield.

unforeseen runtime leaks. It is also no longer possible to abuse certain variables to walk the stack or access the "caller" from within a function.

Finally, strict mode forbids the use of some keywords that will be used in future ECMAScript versions, such as "private", "public", "protected", "interface", ...

Research in the area of JavaScript subsets and rewriting systems includes BrowserShield [RDW+06], CoreScript [YCIS07], ADsafe [Cro], Facebook JavaScript [The], Caja [MSL+08], Jacaranda [Jac], Microsoft Live Websandbox [Micj], Jigsaw [MF12], Gatekeeper [GL09], Blancura [FWB10], Dojo Secure [Kri], ... The remainder of this section discusses a selection of work on JavaScript subsets and rewriting systems.

5.1.1 BrowserShield

Reis et al. have developed BrowserShield, a dynamic instrumentation system for JavaScript. BrowserShield parses and rewrites HTML and JavaScript in a middlebox, rewriting all function calls, property accesses, constructors and control structures to be relayed through specialized methods of the *bshield* object. A client-side JavaScript library then inserts this *bshield* object, which mediates access to DOM methods and properties according to a policy, into the JavaScript execution environment before any scripts run.

BrowserShield aims at preventing the exploitation of browser vulnerabilities, such as MS04-40 [Micf], a buffer overflow in the Microsoft Internet Explorer browser caused by overly long "src" and "name" attributes in certain HTML elements. To shield the browser from attacks against these vulnerabilities, BrowserShield rewrites both HTML and JavaScript, transforming them to filter out any detected attacks. BrowserShield does not use a JavaScript subset, because it needs to be able to rewrite any HTML and JavaScript found on the Internet to be effective.

Although sandboxing is not the main goal of BrowserShield, its rewriting mechanism provides all the necessary machinery to accomplish this goal by tuning the policy. For instance, BrowserShield could have a policy in place to mediate access to the sensitive eval() function. Listing 5.5 shows the output of BrowserShield's rewriting mechanism on a JavaScript example using the eval() function. After the rewriting step, any call to eval() in the original code is relayed through the "bshield" object, which can mediate access at runtime.

A prototype of BrowserShield was implemented as a Microsoft ISA Server 2004 [Mice] plugin for evaluation. The plugin in this server-side middlebox is responsible for rewriting HTML and script elements, and injecting the BrowserShield client-side JavaScript library which implements the "bshield" object and redirects all JavaScript functionality through it. BrowserShield worked as expected during evaluation. The performance evaluation indicated a maximum slowdown of 136x on micro-benchmarks, and on average 2.7x slowdown on rendering a webpage.

5.1.2 ADsafe

The ADsafe subset, developed by Douglas Crockford, is a JavaScript subset designed to allow direct placement of advertisements on webpages in a safe way, while enforcing good coding practices. It removes a number of unsafe JavaScript features and does not allow uncontrolled access to unsafe browser components.

Examples of the removed unsafe JavaScript features are: the use of global variables, the use of "this", eval(), "with", using dangerous object properties like "caller" and "prototype". ADsafe also does not allow the use of the subscript operator, except when it can be verified that the subscript is numerical, e.g. a[i] is not allowed but a[+i] is allowed because "+i" will always produce a number. In addition, ADsafe removes all sources of non-determinism such as "Date" and "Math.random()".

To make use of ADsafe, widgets must be loaded and executed via the "ADSAFE.go()" method. These widgets must adhere to the ADsafe subset, although there is no verification built into ADsafe. Instead, it is recommended to verify subset adherence in any stage of the deployment pipeline with e.g. JSLint [jsla], a JavaScript code quality verification tool.

ADsafe does not allow JavaScript code to make use of the DOM directly. Instead, ADsafe makes a "dom" object available which provides and mediates access to the DOM.

No performance evaluation has been published about ADsafe by its author,

```
1 // original code
2 (function() { return this; })();

3 // code rewritten by FBJS
4 (function() { return ref(this); })();
```

Listing 5.6: Example JavaScript code making use of "this" semantics to return the global object and the code rewritten by FBJS to prevent FBJS code from breaking out of its namespace.

who claim that ADsafe "will not make scripts bigger or slower or alter their behavior" [Cro]. This claim applies if advertisement scripts are written in the ADsafe subset directly, and not translated from full JavaScript.

Research on ADsafe has revealed several problems and vulnerabilities, which allow leaking the document object [TEM+11], launch a XSS attack [FWB10], allow the guest to access properties on the host page's global object [PEGK11], prototype poisoning [MT09] and more.

5.1.3 Facebook JavaScript

Facebook JavaScript (FBJS) is a subset of JavaScript and part of the Facebook Markup Language (FBML) which was used to publish third-party Facebook applications on the Facebook servers. FBJS was designed to allow web application developers as much flexibility as possible while at the same time protecting site integrity and the privacy of Facebook's users.

The FBJS subset excludes some of JavaScript's dangerous constructs such as "eval", "with", "___parent___", "constructor" and "valueOf". A preprocessor rewrites FBJS code so that all top-level identifiers in the code are prefixed with an application-specific prefix, thus isolating the code in its own namespace.

Special care is also taken with e.g. the use of "this" and object indexing to retrieve properties, making sure that a Facebook application cannot break out of its namespace. The semantics of "this" are dependent on the way and location that it is used. A code fragment such as the one listed in Listing 5.6 can return the global object, allowing FBJS code to break out of its namespace. To remedy this problem, the FBJS rewriter encloses all references to "this" with the function "ref()", e.g. "ref(this)". This "ref()" function verifies the way in which it is called at runtime, and prevent FBJS code from breaking out of its

namespace. Similarly, the FBJS rewriter also encloses object indices such as "property" in "object["property"]" with "idx("property")" to also prevent that "this" is bound to the global object.

Research on FBJS has revealed some vulnerabilities [MT09, MMT09], which were addressed by the Facebook team.

Maffeis et al. [MT09] discovered that a specially crafted function can retrieve the current scope object through JavaScript's exception handling mechanism, allowing the "ref()" and "idx()" functions to be redefined. This redefinition in turn allows a FBJS code to break out of its namespace and take over the webpage.

After Facebook fixed the previous issues, Maffeis et al. [MMT09] discovered another vulnerability which allows the global object to be returned on some browsers, by tricking the fixed "idx()" function to return an otherwise hidden property, through a time-of-check-time-of-use vulnerability [MIT].

5.1.4 Caja

Google's Caja, short for Capabilities Attenuate JavaScript Authority, is a JavaScript subset and rewriting system using a server-side middlebox. Caja represents an object-capability safe subset of JavaScript, meaning that any code conforming to this subset can only cause effects outside itself if it is given references to other objects. In Caja, objects have no powerful references to other objects by default and can only be granted new references from the outside. The capability of affecting the outside world is thus reflected by holding a reference to an object in that outside world.

The Caja subset removes some dangerous features from the JavaScript language, such as "with" and "eval()". Furthermore, Caja does not allow variables or properties with names ending in "___" (double-underscore), while at the same time marking variables and properties with names ending in "_" as private.

```
1 window.alert("hello world");
```

Listing 5.7: Example JavaScript code to be cajoled by Caja.

Caja's rewriting mechanism, known as the "cajoler," examines the guest code to determine any free variables and wraps the guest code into a function without

```
1 var tamedwindow = tame(window);
2 var cajoledcode = function(param) {
3    param.alert("hello world");
4 };

5 cajoledcode(tamedwindow);
```

Listing 5.8: Conceptual cajoled code and tamed window.

free variables. Listing 5.7 shows some example code and its cajoled form is shown in Listing 5.8 (the "cajoledcode" variable). In addition, Caja adds inline checks to make sure that Caja's invariants are not broken and that no object references are leaked. The output of the cajoler is cajoled code, which is sent to a client's browser.

On the client-side, objects from the host webpage are "tamed" so that they only expose desired properties before being passed to the cajoled guest code. These tamed objects with carefully exposed properties are the only references that cajoled code obtains to the host page. In this way, all accesses to the DOM can be mediated by taming the global object before passing it to cajoled code. Listing 5.8 shows how the "window" object is tamed and passed to the cajoled form of Listing 5.7.

5.1.5 Discussion

The JavaScript language makes static code verification difficult, because of its dynamic nature (e.g. "eval()") and strange semantics (e.g. the "with" construct). JavaScript subsets eliminate some of JavaScript's language constructs so that code may be more easily verified. When required, JavaScript rewriting systems can transform the code so that policies can also be enforced at runtime.

This section discussed four JavaScript subsets and rewriting mechanisms: BrowserShield, ADsafe, Facebook JavaScript and Caja. Some of their features are summarized in Table 5.1.

It is noteworthy that all three JavaScript subsets remove "with" and "eval()" from the language, which is in line with the standardized JavaScript strict mode subset. The only available performance benchmarks are for BrowserShield, which rewrites code written in full JavaScript, and indicate a heavy performance penalty when rewriting JavaScript in a middlebox. Furthermore, the list of

Table 5.1: Comparison between prominent JavaScript sandboxing systems using subsets and rewriting systems.

System	Target application	Rewrites	Uses subset	Removed features	Performance	Known weaknesses
BrowserShield	Preventing browser exploitation	Y	N	n/a	max. 136x slowdown on micro-benchmarks, avg. 2.7x slowdown on user experience	
ADsafe	Advertising	N	Y	“eval()”, “with”, “this”, global vars, ...	no slowdown	[TEM+11] [FWB10] [PEGK11] [MT09]
FBJS	Third-party widgets	Y	Y	“eval()”, “with”, ...	no data	[MT09] [MMT09]
Caja	Third-party widgets	Y	Y	“eval()”, “with”, ...	no data	

known weaknesses suggest that creating a secure JavaScript subset, although possible, is not an easy task.

JavaScript subsets and code rewriting have been used in real world web applications and have proved to be effective in restricting available functionality to selected pieces of JavaScript code. However, restricting the integration of third-party JavaScript code which conforms to a specific JavaScript subset, puts limitations on third-party JavaScript library developers which they are unlikely to follow without incentive. Even if these developers are willing to limit themselves to a JavaScript subset, they would need to create a version of their code for every subset that they need to conform too. For instance, the jQuery developers would need to create a specific version for use with FBJS, Caja, ADsafe etc. This is an unrealistic expectation.

The standardization of a JavaScript subset, such as e.g. strict mode, helps eliminate this disadvantage for third-party JavaScript providers. But even with a standardized JavaScript subset to aid with code verification, this verification step itself must still happen in a middlebox located at either the server-side or the client-side.

Opting for a middlebox on a server-side has the disadvantage that it changes the architecture of the Internet. From the browser's perspective, JavaScript code would need to be requested from the middlebox instead of directly downloading it from the third-party script provider. Although this poses no problem for generic JavaScript libraries such as jQuery, it does pose a problem for JavaScript code which is generated dynamically depending on the user's credentials, as is the case with e.g. JSONP. In the latter case the third-party script provider might require session information to prove a user's identity, which will not be provided by the browser when requesting said script from a server-side middlebox.

A client-side middlebox on the other hand, does not suffer from this particular problem because it has the option of letting the browser connect to it transparently, e.g. in case of a web proxy. With a client-side middlebox, the web application developers lose control over the rewriting process. Users of the web application should setup the middlebox on the client-side in order to make use of this web application. But requiring users to install a middlebox next to their browser for a single web application, hurts usability and puts a burden on users which they might not like to carry.

From a usability viewpoint, it makes more sense to require only a single middlebox which can be reused for multiple web applications and to integrate this client-side middlebox into the browser somehow.

```
1 var x = ...;
2 window.setTimeout(x, 1000);
```

Listing 5.9: Example JavaScript calling setTimeout() with unknown input.

5.2 JavaScript sandboxing using browser modifications

The previous section showed that JavaScript contains several language constructs that cannot easily be verified to be harmless before executing JavaScript code. Instead of verifying the code beforehand, another approach is to control the execution of JavaScript at runtime and monitor the effect of the executing JavaScript to make sure no harm is done.

In a typical modular browser architecture of a browser, as explained in Section 2.2.1, the JavaScript environment is disconnected from other browser components. These other components, such as the DOM, the network layer, the rendering pipeline or HTML parser are not directly accessible to JavaScript code running in the JavaScript environment. Without these components, JavaScript is effectively side-effect free and is unable to affect the outside world.

The connection layer between the JavaScript engine and the different browser components, is an excellent location to mediate access to the powerful functionality that these components can provide. In order to enforce a policy at this location, the browser must be modified with a mechanism that can intercept, modify and block messages between the JavaScript engine and the different components.

Example: allowing only Function object parameters for setTimeout()

Consider the example in Listing 5.9. In this example, the DOM API function "setTimeout()" is called with a parameter "x." The specification for the "setTimeout()" function in the Web application API standard [WHA] lists two versions: a version where "x" must be a Function object, and a version that allows it to be a String. Passing a string to the "setTimeout()" function is regarded as a bad coding practice and considered as evil as using "eval()" [JSLb]. Because of the inherent difficulty in verifying JavaScript code before runtime, it can be

desirable to enforce a policy at runtime which rejects calls to "setTimeout()" when a string is passed as an argument.

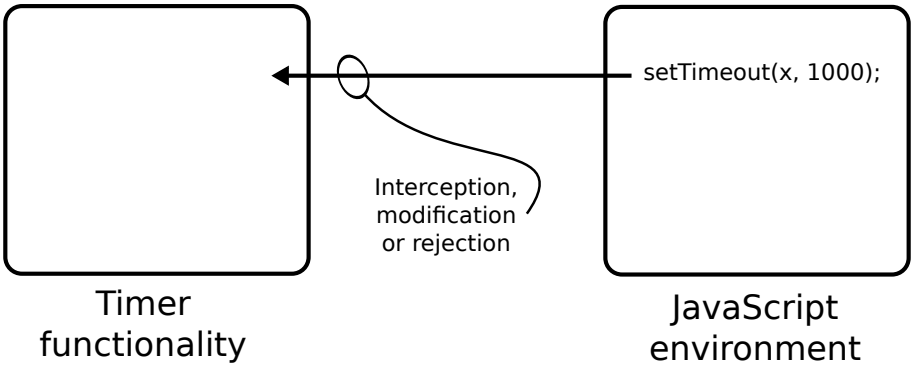


Figure 5.4: Executing the "setTimeout()" function will send a message from the JavaScript environment to the component implementing timer functionality, which can be intercepted, modified or rejected by a policy enforcement mechanism in a modified browser.

The "setTimeout()" function is provided by a browser component which implements timer functionality. To access this function, the JavaScript engine must send a message to this component to invoke the timer functionality, as shown in Figure 5.4. At this point, a browser modified with a suitable policy enforcement mechanism can intercept the message, and reject it if the given parameter is not a Function object.

Forms of browser modifications

Browser modifications can take many forms, but they can generally be split into three groups: browser plugins, browser extensions and browser core modifications.

As discussed in Sections 2.2.2 and 2.2.5, browser plugins and browser extensions can add extra functionality to the browser that can be used to enforce a JavaScript sandboxing technique. They are however limited in the modifications they can make in the browser environment.

For more advanced modifications to the browser, such as e.g. the JavaScript engine or the HTML parser, it is typically the case that neither plugins nor extensions are suitable. Therefor, modifying the browser core itself is required.

```
if (window.JSSecurity) {
  JSSecurity.afterParseHook =
    function(code, elt) {
      if (whitelist[SHA1(code)]) return true;
      else return false;
    };
  whitelist = new Object();
  whitelist["478zB3KkS+UnP2xz8x62ugOxvd4="] = 1;
  whitelist["AOOq/aTVjJ7EWQIsGVeKfdg4Gdo="] = 1;
  ... etc. ...
}
```

Listing 5.10: Example whitelist policy implemented in BEEP's *afterParseHook* function, from [JSH07].

Research on JavaScript sandboxing through some form of browser modification, includes BEEP [JSH07], ConScript [ML10], WebJail [ARD+11], Contego [LD11], AdSentry [DTLJ11], JCShadow [PDL+11], Escudo [JDRC10], ...

5.2.1 Browser-Enforced Embedded Policies (BEEP)

Jim et al. introduce Browser-Enforced Embedded Policies, a browser modification that introduces a callback mechanism, called every time JavaScript is about to be executed. The callback mechanism provides a hook named *afterParseHook* inside the JavaScript environment, which can be overridden by the web developer.

Every time a piece of JavaScript is to be executed, the browser calls the *afterParseHook* callback to determine whether the piece of JavaScript is allowed to execute or not. To be effective, BEEP must be the first JavaScript code to load in the JavaScript environment, in order to set up the *afterParseHook* callback.

The authors experimented with two types of policies: whitelisting and DOM sandboxing.

In the whitelisting policy approach, illustrated in Listing 5.10, the *afterParseHook* callback function receives the script to be executed, and hashes is with the SHA-1 hashing algorithm. This hash is then compared with a list of hashes for allowed scripts. If the hash is found among this whitelist, the *afterParseHook* callback returns "true" and the script is executed.

```
<div class="noexecute">
  <!-- possibly-malicious content starts here -->
  <script>
      alert("hello world");
  </script>
  <!-- possibly-malicious content ends here -->
</div>
```

Listing 5.11: Example HTML with the "noexecute" attribute to be used with BEEP's DOM sandboxing policy.

```
<div class="noexecute">
  <!-- possibly-malicious content starts here -->
  </div><script>
      alert("hello world");
  </script><div>
  <!-- possibly-malicious content ends here -->
</div>
```

Listing 5.12: A node-splitting attack against the example in Listing 5.11. Notice how the enclosing "div" element with "noexecute" attribute is closed by an attacker-injected closing "div" element.

In the DOM sandboxing policy approach, illustrated in Listing 5.11, HTML elements in the web page are clearly marked with a "noexecute" attribute if they can potentially contain untrusted content such as third-party advertising. When a script is about to be executed, the *afterParseHook* callback function receives both the script and the DOM element from which the execution request came. The *afterParseHook* callback function then walks the DOM tree, starting from the given DOM element and following the references to parent nodes. For each DOM node found in this walk, the callback function checks for the presence of a "noexecute" attribute. If such an attribute is found, the *afterParseHook* callback function returns false, rejecting script execution.

The authors report two problems with this last approach. First, in an attack to which the authors refer to as "node-splitting," an attacker may write HTML code into the webpage, allowing him to break out of the enclosing DOM element on which a "noexecute" attribute is placed. Shown in Listing 5.12, an attacker could easily break out of the DOM sandboxing policy by closing and opening the enclosing "div" tag which has the "noexecute" attribute set, hereby escaping

```
<head>
  <script policy='
    let httpOnly: K -> K = function(_ : K) {
      curse(); throw "HTTP-only cookies"; };
    around(getField(document, "cookie"), httpOnly);
    around(setField(document, "cookie"), httpOnly);
    '>
  </script>
</head>
```

Listing 5.13: Example HttpOnly cookie policy defined on a script element using ConScript, adapted from ConScript [ML10].

its associated policy of rejecting untrusted scripts. Second, an attacker can introduce an HTML frame, which creates a child document. The *afterParseHook* callback function inside this child document would not be easily able to walk up the parent's DOM tree to check for "noexecute" attributes.

BEEP was implemented in the Konqueror and Safari browsers, and partially in Opera and Firefox. Performance evaluation indicates an average of 8.3% and 25.7% overhead on the loadtime of typical webpages for a whitelist policy and DOM sandboxing policy respectively.

5.2.2 ConScript

Meyerovich et al. present ConScript, a client-side advice implementation for Microsoft Internet Explorer 8. ConScript allows a web developer to wrap a function with an advice function using *around advice*. The advice function is registered in the JavaScript engine as *deep advice* so that it cannot be altered by an attacker.

As with BEEP, ConScript's policy enforcement mechanism must be configured before any untrusted code gains access to the JavaScript execution environment. ConScript introduces a new attribute "policy" to the HTML <script> tag, in which a web developer can store a policy to be enforced in the current JavaScript environment. When the web page is loaded, ConScript parses this "policy" attribute and registers the contained policy.

Unlike shallow advice, which is within reach of attackers and must be secured in order to prevent tampering by an attacker, ConScript registers the advice function as "deep advice" inside the browser core, out of reach of any potential attacker.

Listing 5.13 shows a ConScript policy being defined in the head of a web page. The policy in this particular example enforces the usage of "HttpOnly" [Mich] cookies, a version of HTTP cookies which cannot be accessed by JavaScript. To achieve this goal, the policy defines a function "HttpOnly" which simply throws an exception, and registers this function as "around" advice on the getter and setter of the "cookie" property of the "document" object, from which regular cookies are accessible in JavaScript.

Using *around advice* as an advice function allows a policy writer full freedom to block or allow a call to an advised function, possibly basing the decision on arguments passed to the advised function at runtime.

ConScript uses a ML-like subset of JavaScript with labeled types and formal inference rules as its policy language, which can be statically verified for common security holes. To showcase the power of ConScript and its policy language, the authors define 17 example policies addressing a variety of observed bugs and anti-patterns, such as: disallowing inline scripts, restricting XMLHttpRequests to encrypted connections, disallowing cookies to be leaked through hyperlinks, limiting popups and more.

ConScript was implemented in Microsoft Internet Explorer 8 and its performance evaluated. On average, ConScript introduces a slowdown during micro-benchmarks of 3.42x and 1.24x after optimizations. The macro-benchmarks are reported to have negligible overhead.

5.2.3 WebJail

We propose WebJail (See Chapter 6), a JavaScript sandboxing mechanism which uses deep advice functions like ConScript.

In WebJail, HTML iframe elements are used as the basis for a sandbox. A new "policy" attribute for an iframe element allows a web developer to specify the URL of a WebJail policy, separating concerns between web developers and policy makers.

We argue that an expressive policy language such as ConScript's can cause confusion with the integrators who need to write the policy, thus slowing the adoption rate of a sandboxing mechanism. In addition, they warn for a scenario dubbed "inverse sandbox," in which the policy language is so expressive that an attacker may use it to attack a target web application by sandboxing it with a well-crafted policy. For instance, if the policy language is the JavaScript language, an attacker may define a policy on an iframe to intercept any cookie-access and transmit these cookies to an attacker-controlled host. A target

```
{
  "framecomm": "yes",
  "extcomm": ["google.com", "youtube.com"],
  "device": "no"
}
```

Listing 5.14: Example WebJail policy allowing inter-frame communication, external communication to Google and YouTube, but disallowing access to the Device API, from [ARD+11].

web-application could then be loaded into this iframe and would, upon accessing its own cookies, trigger the policy mechanism which leaks the cookies to the attacker.

To avoid this scenario, WebJail abstracts away from an overly expressive policy language and defines its own secure composition policy language. Based on an analysis of sensitive JavaScript APIs in the HTML5 specifications, we divided the APIs into nine categories. The policy consists of a file written in JSON, describing access rights for each of these categories. Access to a category of sensitive JavaScript APIs in WebJail can be granted or rejected with "yes" or "no," or determined based on a whitelist of allowed parameters. Listing 5.14 shows an example WebJail policy which allows inter-frame communication (framecomm: yes), external communication to Google and YouTube (extcomm: ["google.com", "youtube.com"]), but disallowing access to the Device API (device: no).

WebJail's architecture, depicted in Figure 5.5 consists of three layers to process an integrator's policy and turn it into deep advice. The policy layer reads an iframe's policy and combines with the policies of any enclosing iframes. Policy composition is an essential step to ensure that an attacker cannot easily escape the sandbox by creating a child document without a policy defined on it. The advice construction layer processes the composed policy and creates advice functions for all functions in the specified JavaScript APIs. Finally, the deep aspect weaving layer combines the advice functions with the API functions, turning them into deep advice and locking them safely inside the JavaScript engine.

WebJail was implemented in Mozilla Firefox 4.0b10pre for evaluation. The performance evaluation indicated an average of between 6.4% and 27% for micro-benchmarks and an average of 6 ms loadtime overhead for macro-benchmarks.

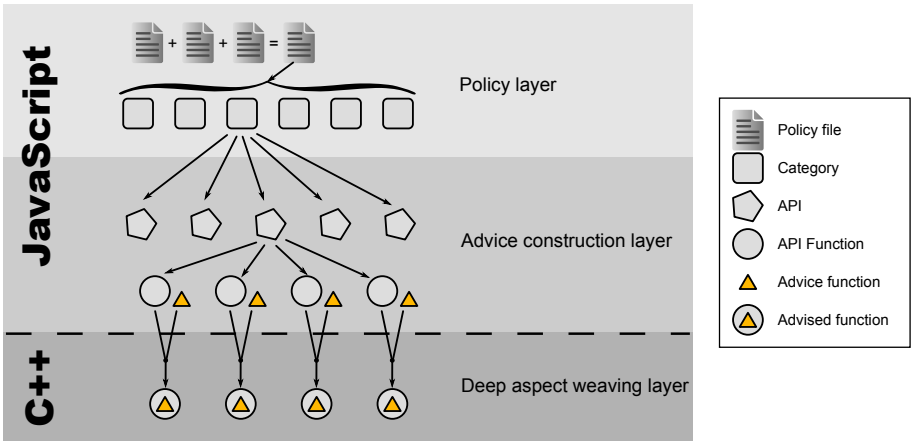


Figure 5.5: The WebJail architecture consists of three layers: the policy layer, the advice construction layer and the deep aspect weaving layer, from [ARD+11].

5.2.4 Contego

Luo et al. design and implement Contego, a capability-based access control system for browsers.

In a capability-based access control model, the ability of a principal to perform an action is called a capability. Without the required capability, the principal cannot perform the associated action.

Contego's authors identified a list of capabilities in browsers, among which: performing Ajax requests, using cookies, making HTTP GET requests, clicking on hyperlinks, ... They list three types of actions that can be associated with those capabilities, based on where they originate: HTML-induced actions, JavaScript-induced actions and event-driven actions.

Contego allows a web developer to assign capabilities to elements in the DOM tree, by assigning a bit-string to the "cap" attribute. Each bit in the bit-string indicates whether a certain capability should be enabled ("1") or disabled ("0") for all DOM elements enclosed by the element on which the capabilities apply. The meaning of each bit in the bit-string is shown in Listing 5.15, which also shows an example policy disabling access to cookies.

The authors warn about a node-splitting attack when an attacker is allowed to insert content into a element. Just as with BEEP's DOM sandboxing policy, care should be taken to avoid that an attacker can insert a closing tag and escape the policy. In addition, Contego has measures in place to ensure that

```
<div cap="1100011111"> ... </div>
<!--
  Capability bitstring:
    1 AJAX POST request allowed
    1 AJAX GET request allowed
    0 Cookie setting not allowed
    0 Cookie getting not allowed
    0 Cookie using not allowed
    1 HTTP GET request allowed
    1 HTTP POST request allowed
    1 Hyperlink click allowed
    1 Button submit click allowed
-->
```

Listing 5.15: Example usage of Contego and its capability bitstring, from [LD11].

an attacker cannot override capability restrictions by e.g. setting a new "cap" attribute either in HTML or in JavaScript. Cases where principals with different capabilities interact are handled by restricting the actions to the conjunction of the capability sets.

To implement Contego in the Google Chrome browser, the authors extended the browser with two new components: the binding system and the enforcement system. The binding system assigns and tracks individual principal's capabilities within a webpage. The enforcement system then uses the information from the binding system to allow or deny actions at runtime.

The performance evaluation shows an average overhead of about 3% on macro-benchmarks.

5.2.5 AdSentry

Dong et al. propose AdSentry, a confinement solution for JavaScript-based advertisements, which executes the advertisements in a special-purpose JavaScript engine.

An architectural overview of AdSentry is shown in Figure 5.6. Next to the regular JavaScript engine, AdSentry implements an additional JavaScript engine, called the shadow JavaScript engine, as a browser plugin. The browser plugin is built on top of the Native Client (NaCl) [Goof] sandbox, which protects the browser and the rest of the operating system from drive-by-download attacks occurring inside the sandbox.

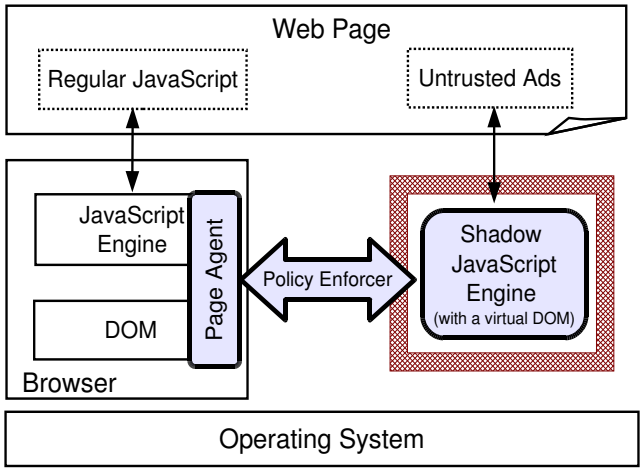


Figure 5.6: The AdSentry architecture: advertisements are executed in a shadow JavaScript engine which communicates with the Page Agent via the policy enforcer, from [DTLJ11].

```
msg ::= command data
command ::= script | callFunc | getProp
                | setProp | return
data ::= <text>
```

Listing 5.16: Format of the communication protocol used between AdSentry's shadow JavaScript engine and the Page Agent, from [DTLJ11].

Advertisements can either be explicitly marked for use with AdSentry, or they can be automatically detected by AdBlock Plus. When an advertisement is detected in a webpage, AdSentry assigns it a unique identifier and communicates with the shadow JavaScript engine to request that the code be executed there. The shadow JavaScript engine then creates a new JavaScript context with its own global object and virtual DOM and executes the advertisement.

The virtual DOM inside the shadow JavaScript context has no access to the real webpage on which the advertisement is supposed to be rendered. Instead, the methods of the virtual DOM are stubs which trigger the shadow JavaScript engine to communicate with a Page Agent in the real JavaScript engine, requesting access on behalf of the advertisement. The communication between the Page Agent and the shadow JavaScript engine is facilitated with a

data exchange protocol, shown in Listing 5.16. This communication channel is also where AdSentry's enforcement mechanism operates, granting or blocking access to the real webpage's DOM according to a user-specified policy. No information is given on how this policy can be specified.

AdSentry was implemented in Google Chrome, and uses a standalone version of SpiderMonkey, Mozilla's JavaScript engine, as the shadow JavaScript engine. The performance evaluation indicates an average overhead of 590x on micro-benchmarks when traversing the boundary between the shadow JavaScript engine and the Page Agent, and an around 3% to 5% overall loadtime overhead on macro-benchmarks.

5.2.6 Discussion

This section discussed five browser modifications that aim to isolate and restrict JavaScript code in the web browser: BEEP, ConScript, WebJail, Contego and AdSentry. Some of their features are summarized in Table 5.2.

JavaScript sandboxing through a browser modification allows the integration of third-party scripts written in the full JavaScript language. Web applications can be built with a much richer set of JavaScript libraries, since those JavaScript libraries are not confined to a subset of JavaScript.

In addition, a browser modification can control the execution of JavaScript inside the browser, allowing the construction of efficient custom-built machinery to enforce a sandboxing policy, ensuring low overhead.

However, modified browsers pose a problem with regard to dissemination of the software and compatibility with browsers and browser versions. End-users must take extra steps in order to enjoy the protection of this type of JavaScript sandboxing systems.

Because end-users do not all use the same browser, it becomes impossible to assure that all end-users can keep using their own favorite browser. In the most fortunate case, the developers of this browser core modification may find a way to port their sandboxing system to all browsers. Even if this is the case, a browser core modification is a fork in a browser's code base and must be maintained to keep up with changes in the main code base, which can be a significant time investment.

Likewise, a browser plugin or extension implementing a certain JavaScript sandboxing system, must also be created for all browser vendors and versions, to enable a wide range of users to make use of it. Such a plugin or extension must

Table 5.2: Comparison between prominent JavaScript sandboxing systems using a browser modification.

System	Target application	Isolation unit	Restricts	Policy expressiveness	Deployment	Browser	Performance	Known weaknesses
BEEP	restrict scripts	entire JS environment	execution of JS scripts	full JavaScript to indicate "accept" or "reject"	*afterParseHook* implementation by integrator	Konqueror, Safari, partially Opera, partially Firefox	8.3% to 25.7% macro	node-splitting
ConScript	sandboxing	entire JS environment	??? anything	high: own JS subset	"policy" attribute on script element	MSIE	1.24x to 3.42x micro, negligible macro	
WebJail	sandboxing	entire JS environment + subframes	access to sensitive APIs	yes/no/whitelist	"policy" attribute on iframe element	Firefox	6.4% to 27% micro, 6 ms macro	
Contego	restrict capabilities	element	capabilities	bitstring	"cap" attribute on div element	Chrome	3% macro	
AdSentry	advertisement	shadow JavaScript engine	access to the DOM	???	???	Chrome	590x micro, 3% to 5% macro	

equally be maintained for future releases of browsers, which can also require a significant time investment.

All in all, modifying a browser through a fork of browser code, a browser plugin or a browser extension in order to implement a JavaScript sandboxing system, is acceptable for a prototype, but proves difficult in a production environment.

An alternative approach is to convince major browser vendors to implement the browser modification as part of their main code base, or even better, pass it through the standardization process so that all browser vendors will implement it. This approach will ensure that the sandboxing technique ends up in a user's favorite browser automatically and that the code base is maintained by the browser vendors themselves.

Unfortunately, getting a proposal accepted by the standardization committees is not a straightforward task, partly because no solution is widely accepted as being "The Solution."

In recent years, the standardization process has yielded new and powerful functionality that could be used to build a JavaScript sandboxing system. Through this approach, a JavaScript sandboxing system would not need any browser modification at all and work out of the box on all browsers that support the latest Web standards.

5.3 JavaScript sandboxing without browser modifications

The previous section showed that a sandboxing mechanism implemented as a browser modification, can be used to restrict JavaScript functionality available to untrusted code at runtime. A browser modification is useful for proof-of-concept evaluation of a sandboxing mechanism, but proves problematic in a production environment. Not only must a browser modification be maintained with new releases of the browser on which it is based, but end-users must also be convinced to install the modified browser, plugin or extension.

Given the powerful nature of JavaScript, it is possible to isolate and restrict untrusted JavaScript code at runtime, without the need for a browser modification. This approach is challenging because the enforcement mechanism will execute in the same execution environment as the untrusted code it is trying to restrict. Special care must be taken to ensure that the untrusted code cannot interfere with the enforcement mechanism, and this without any added functionality to protect itself from the untrusted code.

Isolation unit and communication channel

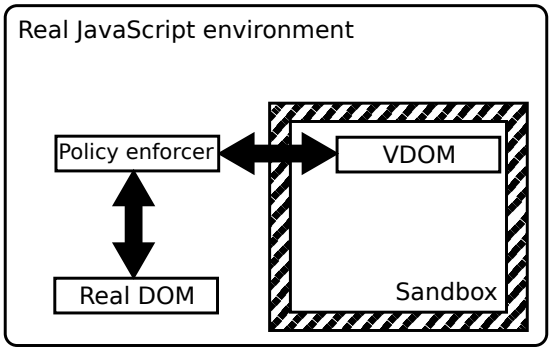


Figure 5.7: Relationship between the real JavaScript environment and a sandbox. The sandbox can only interact with a Virtual DOM, which forwards it via the policy enforcer to the real DOM.

Following the same rationale as in the previous section, a good approach is to create an isolated unit (or sandbox) which is completely cut off from any sensitive functionality, reducing it to a side-effect free execution environment. Figure 5.7 sketches the relationship between a sandbox and the real JavaScript environment.

Any untrusted code executed in the sandbox, will not be able to affect the outside world, except through a virtual DOM introduced into this sandbox. To access the outside world, the isolated code must make use of the virtual DOM, which will forward the access request over a communication channel to an enforcement mechanism. If the access is allowed, the enforcement mechanism again forwards the access request to the real JavaScript environment.

New and powerful ECMAScript 5 functionality

The rise of Web 2.0 resulted in the standardization of ECMAScript 5, which brought new and powerful functionality to mainstream browsers. This new functionality can help with the isolation and restriction of untrusted JavaScript code.

An example of such functionality is the WebWorker API, or WebWorkers [**?**]. WebWorkers allow web developers to spawn background workers to run in parallel with a web page. These workers are intended to perform long-running computational tasks in the background, while keeping web pages responsive to user interaction.

WebWorkers have a very restricted API available to them, which only allows them to do very basic tasks such as set timers, perform XMLHttpRequests or communicate through "postMessage()". In particular, WebWorkers have no access to the DOM. Communication between WebWorkers and a web page is achieved through the postMessage API.

Having new ECMAScript 5 functionality in place in browsers today, opens new options for JavaScript sandboxing mechanisms which previously required browser modifications or code verification/transformation in a separate middlebox.

For instance, because WebWorkers restrict JavaScript code from accessing the DOM and other sensitive JavaScript functionality, they can be used as the isolation unit for a JavaScript sandboxing mechanism. TreeHouse, discussed farther in this section, uses WebWorkers as its isolation unit.

Research on JavaScript sandboxing without browser modification includes Self-protecting JavaScript [PSC09, MPS10], AdJail [LGV10], Object Views [MFM10], JSand [AVAB+12], TreeHouse [IW12], SafeScript [TLPKV13], IceShield [HFH11], SafeJS [CDP13], Two-tier sandbox [PD12], Virtual Browser [CLR+12], ... A selection of this work is discussed in the following sections.

5.3.1 Self-Protecting JavaScript

Phung et al. propose a solution where DOM API functions are replaced by wrappers which can optionally call the original functions, to which the wrapper has unique access. The wrappers can be used to enforce a policy and, with the ability to store state inside the wrapper function's scope, allow the enforcement of very expressive policies. Access to sensitive DOM properties can also be limited by defining a getter and setter method on them which implements a restricting policy.

An example of how a DOM function is replaced with a wrapper, is shown in Listing 5.17. In this example, a wrapper for the function "alert()" is created with a built-in policy to only allow the function to be called twice. A reference to the original native implementation of "alert()" is kept inside the wrapper's scope chain, making it only accessible by the wrapper itself. Finally, the original "alert()" function is replaced by the wrapper.

It is vital that the wrappers are created and put in place of the original DOM functions before any other JavaScript runs inside the JavaScript environment, to achieve full mediation. If any untrusted JavaScript code is run before the wrappers are in place, an attacker may keep copies of the original DOM functions around, thus bypassing any policies that are placed on them later.

```
var wrapper = (function (original) {
    // counter keeps state across
    // multiple function calls
    var counter = 0;

    // create and return the wrapper
    return function(m) {
        if(counter < 2) {
            original(m);
            counter++;
        }
    }
})(window.alert);

window.alert = wrapper;
```

Listing 5.17: Simplified version of Self-protecting JavaScript's creation of a wrapper around the "alert()" function, allowing it to be called maximum twice.

The authors warn that references to DOM functions can also be retrieved through the "contentWindow" property of newly created child documents. To prevent this, access to the "contentWindow" property is denied.

A bug in the "delete" operator of older Firefox browsers also allows overwritten DOM functions to be restored to references to their original native implementations, by simply deleting the wrappers.

A performance evaluation of Self-protecting JavaScript revealed a average of 6.33x slowdown on micro-benchmarks, and a 5.37% average overhead for macro-benchmarks.

Magazinius et al. [MPS10] analyzed Self-protecting JavaScript and uncovered several weaknesses and vulnerabilities that allow the sandboxing mechanism to be bypassed by an attacker.

They note that the original implementation does not remove all references to DOM functions from the JavaScript environment, leaving them open to abuse from attackers. The "alert()" function for instance, has several aliases (such as "window.___proto___.alert"), which must all be replaced with a wrapper for Self-protecting JavaScript to be effective.

Equally, simply denying access to the "contentWindow" property is not sufficient to prevent references to DOM functions from being retrieved from child documents. These references can also be access from child documents through the "frames" property of the "window" object, or from the parent document

through the “parent” property of the “window” object.

They also point out that Self-protecting JavaScript is vulnerable to several types of prototype poisoning attacks, allowing an attacker to get access to the original, unwrapped DOM functions as well as the internal state of a policy wrapper.

Lastly, they remind that an attacker could abuse the caller chain during a wrapper’s execution, by gaining access to the non-standard “caller” property available in functions, allowing an attacker to gain access to the unwrapped DOM functions.

Finally, Magazinius et al. offer solutions to remedy these vulnerabilities by making sure any functions and objects used inside a wrapper are disconnected from the prototype chain to prevent prototype poisoning, and coercing parameters of functions inside wrappers to their expected types in order to further reduce the attack surface.

5.3.2 AdJail

Ter Louw et al. propose AdJail, an advertising framework which enforces JavaScript sandboxing on advertisements.

AdJail allows a web developer to restrict what parts of the web page an advertisement has access to, by marking HTML elements in that web page with the “policy” attribute. This “policy” attribute contains the AdJail policy that is in effect for a certain HTML element and its sub-elements.

The AdJail policy language allows the specification of what HTML elements can be read or written to, and whether that access extends to its sub-elements. The web developer can also define a policy to enable or disable images, Flash or iframes, restrict the size of an advertisement to a certain height and width and allow clicked hyperlinks to open web pages in a new window.

By default, an advertisement is positioned in the “default ad zone,” an HTML element that aids the web developer in positioning the advertisement in the web page. The default policy is set to “deny all.”

An overview of AdJail is shown in Figure 5.8. The advertisement is executed in a “shadow page,” which is a hidden iframe with a different origin, so that it is isolated from the real web page. Those parts of the real web page’s DOM that are marked as readable by the advertisement, are replicated inside the shadow page before the advertisement executes.

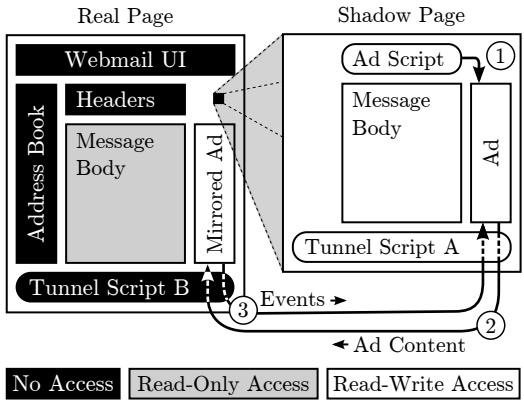


Figure 5.8: Overview of AdJail, showing the real page, the shadow page and the tunnel scripts through which they communicate and on which the policy is enforced, from [LGV10].

Changes made by the advertisement inside the shadow page, are detected by hooking into the DOM of the shadow page, and communicated to the real page through a tunnel script. The changes are replicated on the real page if allowed by the policy. Likewise, events generated by the user on the real page, are communicated to the shadow page so that the advertisement can react to them.

Because AdJail is aimed at sandboxing advertisements, special care must be taken to ensure that the advertisement provider's revenue stream is not tampered with. In particular, AdJail takes special precautions to ensure that content is only downloaded once, to avoid duplicate registration of "ad impressions" on the advertisement network. Furthermore, AdJail leverages techniques used by BLUEPRINT [LV09] to ensure that an advertisement does not inject scripts into the real webpage.

Performance benchmarks indicate that AdJail has an average overhead of 29.7% on ad rendering, increasing the rendering time from an average of 374 ms to 532 ms. Further analysis showed that AdJail has an average overhead of 25% on the entire page loadtime, increasing it from 489 ms to 652 ms.

5.3.3 Object Views

Meyerovich et al. introduce Object Views, a fine grained access control mechanism over shared JavaScript objects.

```
var wrapper = ...;
var obj = { prop: 123, func: function() {
    alert("hello world");
    }
};

defineSetter(wrapper, "prop",
    function(x) {
        obj.prop = x;
    });

defineGetter(wrapper, "prop",
    function() {
        return obj.prop;
    });

wrapper.func = function() {
    obj.func(arguments);
};

alert(wrapper.prop); // displays 123
wrapper.prop = 456;  // sets obj.prop to 456
wrapper.func()       // displays "hello world"
```

Listing 5.18: Pseudo-code showing how an Object View around an object "obj" can be used to intercept reading and writing a property, and intercepting a function call, from [MFM10].

An "Object View" is a wrapper around an object that only exposes a subset of the wrapped object's properties to the outside world. The wrapper consists of a proxy between the wrapped object and the outside world, and a policy that determines what properties should be made available through the proxy.

Sketched in Listing 5.18, an Object View contains a getter and setter method for each property on the wrapped object, and a proxy function for each function object. Writing a value to a property on an Object View, triggers the setter function which may eventually write the value to the wrapped object's respective property. The getter function works in a similar way for reading properties. Using a property of an Object View as a function and calling it, triggers the proxy function. Object Views are applied recursively to a proxy function's return value.

Creating two Object Views that wrap the same object, poses a problem with regard to reference equality. Although comparing the underlying objects of both object views would result in an equality, this would not be the case for the two wrapping Object Views. This inconsistent view can be prevented by only wrapping an object with an Object View once, and returning that same Object

```
{
    "selector": "(//*[@class='example']) | (//*[@class='example']//*)",
    "enabled": true,
    "defaultFieldActions": {read: permit},
    "fields": {shake: {methCall: permit}}
}
```

Listing 5.19: A declarative policy rule specifying that a DOM element of class "example" and its subtree are read-only. If a method "shake" exists, it may be read and invoked as a method.

View every time a new Object View for the underlying object is requested.

Object views offer a basis for fine-grained access control through an aspect system. Each getter, setter and proxy function on an Object View can be combined with an "around" advice function, allowing the enforcement of an expressive policy.

Because of its size and complexity, manually wrapping the entire DOM with object views would be a difficult and error-prone process. Instead, the authors advocate a declarative policy system which is translated into advice for the Object Views.

The declarative policy is specified by a set of rules consisting of an XPath [W3Ck] selector to specify a set of DOM nodes and an Enabled flag to indicate that the selected nodes may be accessed. Optionally, each rule can be extended with default and specific rules for each field of a DOM element. An example rule, shown in Listing 5.19, specifies that all DOM elements of class "example" and its subtree can be accessed (Enabled = true) and is by default read-only (defaultFieldActions). A specific rule for a field called "shake" allows that field to be read and invoked as a method.

The authors discuss using Object Views in two scenarios: a scenario where JavaScript is rewritten[1] to make use of Object Views for same-origin usage, and a scenario where Object Views are used in cross-origin communication between frames.

In the latter scenario, each frame provides an Object View around its enclosed document to only expose the view required by the other. Communication

[1]This work could also be listed under Section 5.1, but since the published paper mostly focuses on the cross-origin communication which does not require browser modifications, it is listed in this section instead.

between the frames is handled by marshaling requests for the other side to a string and transmitting it with "postMessage()". Because each Object View has its own built-in policy, the communication channel does not need to enforce a separate policy.

The performance of Object Views was evaluated on a scenario where several objects are wrapped in a view, but where the communication between Object Views is not marshaled and transmitted with "postMessage()". For this scenario, the average overhead is between 15% and 236% on micro-benchmarks.

5.3.4 JSand

We propose JSand (See Chapter 7), a JavaScript sandboxing mechanism based on Secure ECMAScript (SES).

Secure ECMAScript (SES) is a subset of ECMAScript 5 strict which forms a true object-capability language, guaranteeing that references to objects can only be obtained if they were explicitly passed to an object-capability environment.

Without a reference to the DOM, JavaScript code running in a SES environment cannot affect the outside world. JSand wraps the global object using the Proxy API [ECM] and passes a reference to this proxied global object to the SES environment. Any access to the global object from inside the SES environment, will traverse the proxy wrapper on which a policy can be enforced.

Without additional care, JavaScript inside the SES environment with access to this proxied global object, can invoke methods that return unwrapped JavaScript objects. Such an oversight can cause a reference to the real JavaScript to leak into the SES environment, making JSand ineffective. To avoid this, JSand wraps return values recursively, according to the Membrane Pattern [Mil06]. In addition, JSand preserves pointer equality between wrappers around the same objects, by storing created wrappers in a cache and returning an existing wrapper if one already exists.

Using the Membrane pattern, any access to the outside world from inside the SES environment, can be intercepted and subjected to a policy enforcement mechanism. We do not specify a specific policy implementation, but point out that JSand's architecture allows for expressive fine-grained and stateful policies.

There are two important incompatibilities between the SES subset and ECMAScript 5 code, which makes legacy JavaScript incompatible with JSand.

The first is the mirroring of global variables with properties on the global object and vice versa. When a global variable is created under ECMAScript 5, a

property with the same name is created on the global object. Similarly, a property created on the global object results in the creation of a global variable of the same name. This ECMAScript 5 behavior is not present in SES and can cause legacy scripts who depend on that behavior, to break.

Second, because SES is a subset of ECMAScript 5 strict, it does not support the "with" construct, does not bind "this" to the global object in a function call and does not create new variables during "eval()" invocations. Legacy scripts making use of this behavior will also break in SES.

To be backwards compatible with legacy JavaScript that does not conform to SES, JSand applies a client-side JavaScript rewriting step where needed before sandboxing the guest JavaScript code. The UglifyJS [Mih] JavaScript parser is used to parse JavaScript into an Abstract Syntax Tree (AST). This tree is then inspected and modified for legacy ECMAScript 5 constructs that will break in SES. In particular, JSand rewrites guest code so that the mirroring of global variables and properties of the global object in ECMAScript 5, is replicated explicitly. JSand also finds all occurrences to the "this" keyword and replaces it with an expression that replaces it with "window" if its value is undefined, thus also replicating ECMAScript 5 behavior.

JSand's performance evaluation indicates an average 9x slowdown for function-calls than traverse the membrane wrapper, resulting in an average of 31.2% overhead in user experience when interacting with a realistic web application. The load-time of a web application is increased on average by 365% for legacy web applications using ECMAScript 5 code which requires the rewriting step. We expect that this rewriting step will not be needed in the future, so that the average load-time overhead will drop to 203%.

5.3.5 TreeHouse

Ingram et al. propose TreeHouse, a JavaScript sandboxing mechanism built on WebWorkers. As explained previously, WebWorkers are parallel JavaScript execution environments without a usual DOM, which can only communicate through postMessage.

An overview of TreeHouse's architecture is shown in Figure 5.9. TreeHouse loads guest JavaScript code into a WebWorker to isolate it from the rest of a web page. WebWorkers do not have a regular DOM, so TreeHouse installs a broker with a virtual DOM inside the WebWorker that emulates the DOM of a real webpage. When this virtual DOM is accessed, the broker first consults the policy to determine whether access is allowed. If access is allowed, the broker

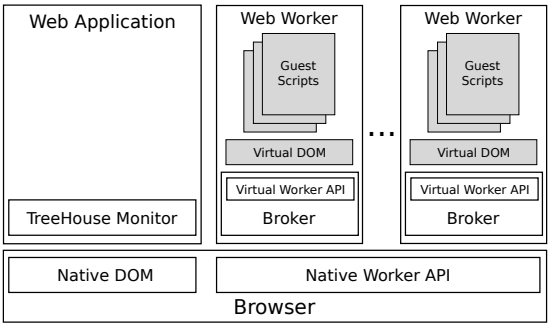


Figure 5.9: TreeHouse architectural overview. Sandboxes consist of WebWorkers with a virtual DOM. Access to this virtual DOM is mediated by broker according to a policy. If access is allowed, the request is forwarded to the real page's monitor, from [IW12].

then forwards the access request to the real page's "TreeHouse Monitor" using "postMessage()," which handles the access to the real page's DOM.

TreeHouse offers two deployment options to web developers wishing to use its sandboxing mechanism. One option is to create a sandbox with a policy and load JavaScript in it manually using the TreeHouse API. Another option, is more user-friendly and allows a web developer to specify guest code to be sandboxed, in actual <script> elements. These <script> elements should have their "type" attribute set to "text/x-treehouse-javascript" to prevent them from being executed by the JavaScript engine in the host page. The special script type is also automatically detected by the TreeHouse Monitor, which will create sandboxes and load the script inside them.

An example use of TreeHouse is shown in Listing 5.20. Here, the first <script> element shows how a sandbox is created called "worker1," with access to the DOM element with id "#tetris" and its subtree. The "tetris.js" script is then loaded inside the sandbox and executed. The second <script> tag references the sandbox "worker1" and indicates through the "data-treehouse-sandbox-policy" attribute that the script "tetris-policy.js" should be interpreted as a policy instead of guest JavaScript code.

A TreeHouse policy consists of a mapping between DOM elements and rules. There are three types of rules: a rule can be expressed by a boolean, a function returning a boolean, or a regular expression. If the rule has a boolean value of True, access to the associated DOM element is allowed. If the rule is a function, that function is invoked at policy enforcement time by the broker, and access is allowed if the return value is True. Finally, if the rule is a regular expression, it

```
<script src="tetris.js"
  type="text/x-treehouse-javascript"
  data-treehouse-sandbox-name="worker1"
  data-treehouse-sandbox-children="#tetris">
</script>
<script src="tetris-policy.js"
  type="text/x-treehouse-javascript"
  data-treehouse-sandbox-name="worker1"
  data-treehouse-sandbox-policy>
</script>
```

Listing 5.20: TreeHouse integration in a web page. Guest code is loaded into <script> tags with type "text/x-treehouse-javascript" so that they are automatically sandboxed. The policy is also specified in a <script> element marked with a "data-treehouse-sandbox-policy" attribute, from [IW12].

refers to a property. If the regular expression matches a property's name, then the guest code is allowed to set a value to that property.

Because WebWorkers are concurrent by design, they present a problem when multiple TreeHouse sandboxes try to access to same DOM element in a real page. Such simultaneous access would cause a race condition and result in undefined behavior. To prevent such a race condition, TreeHouse allows a DOM element to only be accessed by one sandbox.

Another concurrency problem arises when the guest code makes use of a synchronous method such as "window.alert()". The guest code will expect the JavaScript execution to block, waiting for the end-user to click away the pop-up window. In reality, TreeHouse's communication channel between the host page and the WebWorkers is asynchronous because "postMessage()" is asynchronous. When calling "window.alert()" in the guest code, the broker would send an asynchronous message to the host page, and let code execution in the sandbox resume immediately. This conflicts with the guest code's expected behavior. The authors chose not to handle this case and raise a runtime exception when guest code calls synchronous methods.

The performance benchmarks for TreeHouse show an average slowdown of 15x to 176x for macro-benchmarks, and an average of 7x to 8000x slowdown on micro-benchmarks for method invocations on the DOM.

```
<!-- the transformation tool -->
<script src='rewriter.js'></script>
<!-- an API's implementation -->
<script src='interface0.js'></script>

<script>
var namespace0 = $_sm[0]();
var script0_code = load_script('http://3rd.com/main.js');
exec_script(transform(script0_code, namespace0));
</script>
```

Listing 5.21: SafeScript used on a webpage. After loading the rewriter and API implementation, a namespace is created and the guest code is loaded. Afterwards, the guest code is transformed so that the property resolution mechanism is locked to the created namespace, and the transformed code is executed, from [TLPKV13].

5.3.6 SafeScript

Ter Louw et al. propose SafeScript, a client-side JavaScript transformation technique to isolate JavaScript code in namespaces.

SafeScript makes use of Narcissus [Mozf], a JavaScript meta-interpreter, to rewrite JavaScript code on the client-side and instrument the code so that it can interpose on the property resolution mechanism. Narcissus is a full JavaScript interpreter and can correctly handle all of JavaScript's strange semantics, its scoping, prototype chains and thus also the property resolution mechanism.

Through this rewriting step, SafeScript can separate JavaScript code in namespaces by manipulating the property resolution mechanism for each sandboxed script so that it ultimately resolves to its own isolated global object. Because property resolution is under SafeScript's control, it can effectively mediate access to the real DOM when sandboxed JavaScript guest code requests access to it.

Listing 5.21 shows how SafeScript can be used to sandbox a given JavaScript. In this example, the "rewriter.js" script contains SafeScript's transformation code and "interface0.js" contains an API implementation for a "namespace 0." After creating the namespace with "$_sm[0]()", the guest code is loaded from a third-party host, transformed so that the property resolution mechanism is locked to namespace 0, and then executed.

SafeScript ensures that any dynamically generated JavaScript code is also

transformed and isolated in a namespace. In order to do so, SafeScript traps methods such as "eval()", "setTimeout()", which can inject JavaScript code into the execution environment directly. To capture JavaScript code that is indirectly injected, SafeScript monitors methods such as "document.write()" and properties like "innerHTML." HTML written through these injection points must first be parsed and have its JavaScript code extracted before it can be transformed by SafeScript.

Despite its many optimizations, SafeScript's performance benchmarks indicate an average slowdown of 6.43x on basic operations such a variable incrementation, because SafeScript rewrites every variable statement. The macro-benchmark reveals an average slowdown of 64x.

5.3.7 Discussion

This section discussed six JavaScript sandboxing mechanisms that do not require any browser modifications: Self-protecting JavaScript, AdJail, Object Views, JSand, TreeHouse and SafeScript. Some of their features are summarized in Table 5.3.

Besides Self-protecting JavaScript, which protects all access-routes to the DOM API through enumeration, all solutions isolate untrusted JavaScript in an isolation unit. The isolated JavaScript cannot access the DOM directly, but must communicate with the real web page and request access, which is then mediated by a policy enforcement mechanism.

JavaScript sandboxing systems that do not require browser modifications leverage existing standardized powerful functionality that is available in browsers today. The advantage of this approach is that standardized functionality is, or in the near future will be, available in all browsers and thus the sandbox works out of the box for all Internet users.

Much of the new browser functionality incorporated in the previously discussed JavaScript sandboxing systems, was not designed for sandboxing and may not perform well enough for a seamless user experience.

In the future that may change, because browser vendors optimize their code for speed to compete with other browser vendors. When new browser functionality becomes more popular, it will undoubtedly also be optimized for speed, automatically increasing the performance of the JavaScript sandboxing systems making use of it.

Web standards keep evolving, so that we can expect more advanced browser functionality in the future. This new functionality can then be used to design and

Table 5.3: Comparison between prominent JavaScript sandboxing systems not requiring browser modifications.

System	**Target application**	**Isolation unit**	**Communication**	**Policy expressiveness**	**Deployment**	**Performance**	**Known weaknesses**
Self-protecting JavaScript	sandboxing	JavaScript environment	n/a	high	library	6.33x micro, 5.37% macro	[MPS10]
AdJail	advertisements	shadow page	postMessage	read/write elements + enable/disable images/other	"policy" attribute	25% macro	
Object Views	sandboxing	iframe	postMessage	get/set/call	declarative policy with XPath	15% to 236% micro	
JSand	sandboxing	SES environment	Membrane/Proxy API	high	VDOM implementation	9x micro, 203% to 365% macro on loadtime, 31.2% macro on user experience	
TreeHouse	sandboxing	WebWorker	postMessage	high	script elements with custom type	7x to 8000x micro, 15x to 176x macro	
SafeScript	sandboxing	namespace	VDOM implementation	high?	VDOM implementation	6.43x micro, 64x macro	

implement yet more powerful JavaScript sandboxing systems. Ideally, this new functionality will also bring APIs dedicated to JavaScript sandboxing, providing purpose-built mechanisms to isolate code in a sandbox and communicate with that sandbox.

When such specialized JavaScript APIs are adopted and implemented, future JavaScript sandboxing mechanisms will no longer need to rely on repurposed functionality, making them simpler and faster.

5.4 Conclusion

This chapter gave an overview of the JavaScript sandboxing research field and the different approaches taken to isolate and restrict JavaScript to a chosen set of resources and functionality.

The JavaScript sandboxing research can be divided into three categories: JavaScript subsets and rewriting systems, JavaScript sandboxing through browser modifications and JavaScript sandboxing without browser modifications.

JavaScript subsets and rewriting systems can restrict untrusted JavaScript if it adheres to a JavaScript subset, but a middlebox needs to verify that this is the case, possibly rewriting the code. These middleboxes break the architecture of the Web when implemented on the server-side, and put an extra burden on the user if implemented on the client-side.

Browser modifications are powerful and can sandbox JavaScript efficiently, because of their prime access to the JavaScript execution environment. Unfortunately, the software modifications are difficult to distribute and maintain in the long run unless they are adopted by mainstream browser vendors.

JavaScript sandboxing mechanisms without browser modifications leverage existing browser functionality to isolate and restrict JavaScript. This approach can be slower but requires no redistribution and maintenance of browser code. In addition, it automatically works on all modern browsers.

In this work, we prefer not to change the architecture of the Web by introducing middleboxes on the server side. Having a middlebox on the client side is possible, but it would be better if the functionality were integrated into a user's browser. Following this line of reasoning, we opted to experiment with JavaScript sandboxing techniques implemented in the browser.

The following two chapters contain two JavaScript sandboxing techniques: Chapter 6 discusses WebJail, a browser-core modification to enforce fine-grained

sandboxing, and Chapter 7 introduces JSand, a JavaScript sandboxing technique that does not require any browser modifications.

Chapter 6

WebJail: Least-privilege Integration of Third-Party Components in Web Mashups

Publication data

Contained in this chapter is the paper titled "WebJail: Least-privilege Integration of Third-party Components in Web Mashups," as presented at the 2011 Annual Computer Security Applications Conference (ACSAC 2011) [ARD+11]. Steven Van Acker was the lead author of this work.

Preamble

This chapter looks into JavaScript sandboxing by using a browser modification.

This chapter presents WebJail, a client-side security architecture to enable least-privilege separation of components in a web mashup through a JavaScript engine modification.

The standardization of new and powerful HTML5 JavaScript APIs brings new and security-sensitive functionality to browsers that can be abused by malicious or compromised third-party web mashup providers.

WebJail is a policy-based client-side security architecture which isolates web mashup components in iframes with a restricted JavaScript execution environment.

WebJail allows a web mashup integrator to specify which JavaScript functionality categories should be available for a mashup component, using a high-level policy language. The WebJail architecture has three layers: the policy layer, the advice construction layer and the deep aspect weaving layer.

The policy layer faces the web mashup integrator and combines WebJail policies from nested outer iframes into a combined policy for the iframe in which the web mashup component is to be isolated.

The advice construction layer translates the policy into "advice" functions, which mediate access to security-sensitive functionality.

The deep aspect weaving layer stores and locks away these advice functions in the JavaScript engine and binds them to the low-level DOM functions in such a way that the DOM functions can no longer be called directly, but the function call will always be mediated by the advice function.

We built a prototype of WebJail in Firefox for evaluation and applied it successfully to several mainstream mashup scenarios. Benchmarks show little overhead when using WebJail: on average 7 ms overhead for a page load and 0.1 ms for a security sensitive operation.

The main contributions of this research are:

- The design of WebJail, a novel security architecture for JavaScript sandboxing of web mashup components,
- The design of a composable policy language, tuned to support WebJail and restrict access to powerful HTML5 APIs,
- The implementation of WebJail and its policy language in Firefox, and its evaluation.

In hindsight, WebJail was the first JavaScript sandboxing mechanism in a web browser, that used deep advice and had a user-friendly policy-composition language that covered the sensitive new HTML 5 APIs.

WebJail showed that it was possible to implement an efficient and user-friendly JavaScript sandboxing mechanism with a minimal amount of code added to a JavaScript engine such as Firefox's Spidermonkey. Being a browser modification however, also had a drawback.

Browser modifications like WebJail are only effective for web mashup developers if the users of those mashups make use of the modified browser. Distribution of a browser modification to end-users is difficult because they have no incentive to go through the trouble of replacing their favorite browser with one that looks the same on the outside. Most likely, this new browser also has less or no support from the community and no longer enjoys the comfort of automatic software updates that all mainstream browsers supply today.

Issues with browser modifications can be avoided by having that modification adopted by all browser vendors. Such an adoption can be achieved by stepping the modification through the standardization process and have it accepted, which is not an easy task.

Instead of building a JavaScript sandboxing system by introducing changes in the browser, which are unlikely to be adopted by all browser vendors, it is better to build it on top of functionality which is already standardized.

Chapter 7 will discuss JSand, which is such a JavaScript sandboxing system that does not require browser modifications, but instead builds on top of already available browser functionality.

Abstract

In the last decade, the Internet landscape has transformed from a mostly static world into Web 2.0, where the use of web applications and mashups has become a daily routine for many Internet users. Web mashups are web applications that combine data and functionality from several sources or components. Ideally, these components contain benign code from trusted sources. Unfortunately, the reality is very different. Web mashup components can misbehave and perform unwanted actions on behalf of the web mashup's user.

Current mashup integration techniques either impose no restrictions on the execution of a third-party component, or simply rely on the Same-Origin Policy. A least-privilege approach, in which a mashup integrator can restrict the functionality available to each component, cannot be implemented using the current integration techniques, without ownership over the component's code.

We propose WebJail, a novel client-side security architecture to enable least-privilege integration of components into a web mashup, based on high-level policies that restrict the available functionality in each individual component. The policy language was synthesized from a study and categorization of sensitive operations in the upcoming HTML 5 JavaScript APIs, and full mediation is achieved via the use of deep aspects in the browser.

We have implemented a prototype of WebJail in Mozilla Firefox 4.0, and applied it successfully to mainstream platforms such as iGoogle and Facebook. In addition, micro-benchmarks registered a negligible performance penalty for page load-time (7 ms), and the execution overhead in case of sensitive operations (0.1 ms).

6.1 Introduction

The Internet has seen an explosion of dynamic websites in the last decade, not in the least because of the power of JavaScript. With JavaScript, web developers gain the ability to execute code on the client-side, providing for a richer and more interactive web experience. The popularity of JavaScript has increased even more since the advent of Web 2.0.

Web mashups are a prime example of Web 2.0. In a web mashup, data and functionality from multiple stakeholders are combined into a new flexible and lightweight client-side application. By doing so, a mashup generates added value, which is one of the most important incentives behind building mashups. Web mashups depend on collaboration and interaction between the different

mashup components, but the trustworthiness of the service providers delivering components may strongly vary.

The two most wide-spread techniques to integrate third-party components into a mashup are via script inclusion and via (sandboxed) iframe integration, as will be discussed in more detail in Section 6.2. The script inclusion technique implies that the third-party component executes with the same rights as the integrator, whereas the latter technique restricts the execution of the third-party component according to the Same-Origin Policy. More fine-grained techniques (such as Caja [MSL+08] or FBJS [The]) require (some form of) ownership over the code to transform or restrict the component to a known safe subset before delivery to the browser. This makes these techniques less applicable to integrate third-party components directly from their service providers.

To enable the necessary collaboration and interaction while restricting the capabilities of untrusted third-party components, web mashups should integrate components according to the least-privilege principle. This means that each of the components is only granted access to data or functionality necessary to perform its core function. Unfortunately, least-privilege integration of third-party mashup components cannot be achieved with the current script-inclusion and frame-integration techniques. Moreover, the need for least-privilege integration becomes highly relevant, especially because of the augmented capabilities of the upcoming HTML5 JavaScript APIs [W3Ci] (such as access to local storage, geolocation, media capture and cross-domain communication).

In this chapter, we propose WebJail, a novel client-side security architecture to enable the least-privilege integration of third-party components in web mashups. The security restrictions in place are configurable via a high-level composition policy under control of the mashup integrator, and allow the use of legacy mashup components, directly served by multiple service providers.

In summary, the contributions of this chapter are:

1. a novel client-side security architecture, WebJail, that supports least-privilege composition of legacy third-party mashup-components
2. the design of a policy language for WebJail that is tuned to support the effective use of WebJail to limit access to the powerful upcoming HTML5 APIs
3. the implementation of WebJail and its policy language in Firefox, and evaluation and discussion of performance and usability

The rest of this chapter is structured as follows. Section 6.2 sketches the necessary background, and Section 6.3 further elaborates the problem statement.

In Section 6.4, the WebJail least-privilege integration architecture is presented and its three layers are discussed in more detail. Next, the prototype implementation in Firefox is described in Section 6.5, followed by an experimental evaluation in Section 6.6 and discussion in Section 6.7. Finally, Section 6.8 discusses related work, and Section 6.9 summarizes the contributions.

6.2 Background

This section briefly summarizes the Same-Origin Policy. Next, Section 6.2.2 discusses how mashups are constructed and gives some insights in the state-of-practice on how third-party mashup components get integrated.

6.2.1 Same-Origin Policy

Currently, mashup security is based on the de facto security policy of the web: the Same-Origin Policy (SOP) [Zal10]. An origin is a domain name-protocol-port triple, and the SOP states that scripts from one origin should not be able to access content from other origins. This prevents scripts from stealing data, cookies or login credentials from other sites. In addition to the SOP, browsers also apply a frame navigation policy, which restricts the navigation of frames to its descendants [BJM09].

Among others, the Same-Origin Policy allows a per-origin separation of JavaScript execution contexts. Contexts are separated based on the origin of the window's document, possibly relaxed via the `document.domain` property to a right-hand, fully-qualified fragment of its current hostname. Within an execution context, the SOP does not impose any additional security restriction.

6.2.2 Integration of mashup components

The idea behind a web mashup is to integrate several web applications (components) and mash up their code, data and results. The result is a new web application that is more useful than the sum of its parts. Several publicly available web applications [Pro] provide APIs that allow them to be used as third-party components for web mashups.

To build a client-side mashup, an integrator selects the relevant in-house and third-party components, and provides the necessary glue code on an integrating

web page to retrieve the third-party components from their respective service providers and let them interact and collaborate with each other.

As stated before, the two most-widespread techniques to integrate third-party components into a web mashup are through script inclusion or via (sandboxed) iframe-integration [RDD+10, MAS10].

Script inclusion. HTML script tags are used to execute JavaScript while a webpage is loading. This JavaScript code can be located on a different server than the webpage it is executing in. When executing, the browser will treat the code as if it originated from the same origin as the webpage itself, without any restrictions of the Same-Origin Policy.

The included code executes in the same JavaScript context, has access to the code of the integrating webpage and all of its datastructures. All sensitive JavaScript operations available to the integrating webpage are also available to the integrated component.

(Sandboxed) iframe integration. HTML iframe tags allow a web developer to include one document inside another. The integrated document is loaded in its own environment almost as if it were loaded in a separate browser window. The advantage of using an iframe in a mashup is that the integrated component from another origin is isolated from the integrating webpage via the Same-Origin Policy. However, the code running inside of the iframe still has access to all of the same sensitive JavaScript operations as the integrating webpage, albeit limited to its own execution context (i.e. origin). For instance, a third-party component can use local storage APIs, but only has access to the local storage of its own origin.

HTML 5 adds the "sandbox" attribute to the iframe element, allowing an integrator to disable all security-sensitive features through its "allow-scripts" keyword. Obviously, this very coarse-grained control has only a very limited applicability in a web mashup context.

6.3 Problem Statement

In this section, the attacker model is specified, as well as two typical attack vectors. Next, the increasing impact of insecure mashup composition is discussed in the context of the upcoming set of HTML5 specifications. Finally, the security assessment is concluded by identifying the requirements for secure mashup composition, namely the least-privilege integration of third-party mashup components.

6.3.1 Attacker model

Our attacker model is inspired by the definition of a *gadget attacker* in Barth *et al.* [BJM09]. The term gadget in their definition should, in the context of this chapter, be read as "third-party mashup component."

We describe the attacker in scope as follows:

Malicious third-party component provider
: The attacker is a malicious principal owning one or more machines on the network. The attacker is able to trick the integrator in embedding a third-party component under control of the attacker.

We assume a mashup that consists of multiple third-party components from several service providers, and an honest mashup consumer (i.e. end-user). A malicious third-party component provider attempts to steal sensitive data outside its trust boundary (e.g. reading from origin-specific client-side storage), impersonate other third-party components or the integrator (e.g. requesting access to geolocation data on behalf of the integrator) or falsely operate on behalf of the end-user towards the integrator or other service providers (e.g. requesting cross-application content with XMLHttpRequest).

We have identified two possible ways in which an attacker could present himself as a malicious third-party component provider: he could offer a malicious third-party component towards mashup integrators (e.g. via a malicious advertisement, or via a malicious clone of a popular component), or he could hack into an existing third-party component of a service provider and abuse the prior existing trust relationship between the integrator and the service provider.

In this chapter, we consider the mashup integrator as trusted by the mashup consumer (i.e. end-user), and an attacker has no control over the integrator, except for the attacker's ability to embed a third-party components of his choice. In addition, we assume that the attacker has no special network abilities (such as sniffing the network traffic between client and servers), browser abilities (e.g. extension under control of the attacker or client-side malware) and is constrained in the browser by the Same-Origin Policy.

6.3.2 Security-sensitive JavaScript operations

The impact of running arbitrary JavaScript code in an insecure mashup composition is equivalent to acquiring XSS capabilities, either in the context of the component's origin, or in the context of the integrator. For instance, a

malicious third-party component provider can invoke typical security-sensitive operations such as the retrieval of cookies, navigation of the browser to another page, launch of external requests or access and updates to the Document Object Model (DOM).

However, with the emerging HTML5 specification and APIs, the impact of injecting and executing arbitrary JavaScript has massively increased. Recently, JavaScript APIs have been proposed to access geolocation information and system information (such as CPU load and ambient sensors), to capture audio and video, to store and retrieve data from a client-side datastore, to communicate between windows as well as with remote servers.

As a result, executing arbitrary JavaScript becomes much more attractive to attackers, even if the JavaScript execution is restricted to the origin of the component, or a unique origin in case of a sandbox.

6.3.3 Least-privilege integration

Taking into account the attack vectors present in current mashup composition, and the increasing impact of such attacks due to newly-added browser features, there is clearly a need to limit the power of third-party mashup components under control of the attacker.

Optimally, mashup components should be integrated according to the least-privilege principle. This means that each of the components is only granted access to data or functionality necessary to perform its core function. This would enable the necessary collaboration and interaction while restricting the capabilities of untrusted third-party components.

Unfortunately, a least-privilege integration of third-party mashup components cannot be achieved with the current script-inclusion and iframe-integration techniques. These techniques are too coarse-grained: either no restrictions (or only the Same-Origin Policy) are imposed on the execution of a third-party component, implicitly inviting abuse, or JavaScript is fully disabled, preventing any potential abuse but also fully killing desired functionality.

To make sure that attackers described in Section 6.3.1 do not exploit the insecure composition attack vectors and multiply their impact by using the security sensitive HTML5 APIs described in Section 6.3.2, the web platform needs a security architecture that supports least-privilege integration of web components. Since client-side mashups are composed in the browser, this architecture must necessarily be implemented in the browser. It should satisfy the following requirements:

R1 Full mediation.
The security-sensitive operations need to be fully mediated. The attacker cannot circumvent the security mechanisms in place.

R2 Remote component delivery.
The security mechanism must allow the use of legacy third-party components and the direct delivery of components from the service provider to the browser environment.

R3 Secure composition policy.
The secure composition policy must be configurable (and manageable) by the mashup integrator. The policy must allow fine-grained control over a single third-party component, with respect to the security-sensitive operations in the HTML5 APIs.

R4 Performance.
The security mechanism should only introduce a minimal performance penalty, unnoticeable to the end-user.

Existing technologies like e.g. Caja [MSL+08] and FBJS [The] require pre-processing of mashup components, while ConScript [ML10] does not work in a mashup context because it depends on the mashup component to load and enforce its own policy. A more thorough discussion of related work can be found in Section 6.8.

6.4 WebJail Architecture

To enable least-privilege integration of third-party mashup components, we propose WebJail, a novel client-side security architecture. WebJail allows a mashup integrator to apply the least-privilege principle on the individual components of the mashup, by letting the integrator express a secure composition policy and enforce the policy within the browser by building on top of the deep advice approach of ConScript [ML10].

The secure composition policy defines the set of security-sensitive operations that the component is allowed to invoke. Each particular operation can be allowed, disallowed, or restricted to a self-defined whitelist. Once loaded, the deep aspect layer will ensure that the policy is enforced on every accesspath to the security-sensitive operations, and that the policy cannot be tampered with.

The WebJail architecture consists of three abstraction layers as shown in Figure 6.1. The upper layer, the *policy layer*, associates the secure composition policy with a mashup component, and triggers the underlying layers to enforce

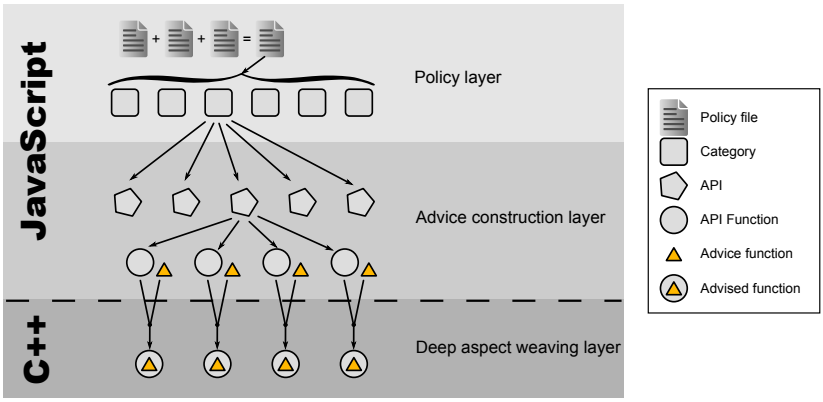


Figure 6.1: The WebJail architecture consists of three layers: The policy layer, the advice construction layer and the deep aspect weaving layer.

the policy for the given component. The lower layer, the *deep aspect weaving layer*, enables the deep aspect support with the browser's JavaScript engine. The *advice construction layer* in between takes care of mapping the higher-level policy blocks onto the low-level security-sensitive operations via a 2-step policy refinement process.

In this section, the three layers of the WebJail will be described in more detail. Next, Section 6.5 will discuss a prototype implementation of this architecture in Mozilla Firefox.

6.4.1 Policy layer

The policy layer associates the secure composition policy with the respective mashup component. In this section, an analysis of security-sensitive operations in the HTML5 APIs is reported and discussed, as well as the secure composition policy itself.

Security-sensitive JavaScript operations

As part of this research, we have analyzed the emerging specifications and browser implementations, and have identified 86 security-sensitive operations, accessible via JavaScript APIs. We have synthesized the newly-added features of these specifications in Figure 6.2, and we will briefly summarize each of the

components in the next paragraphs. Most of these features rely on (some form of) user-consent and/or have origin-restrictions in place.

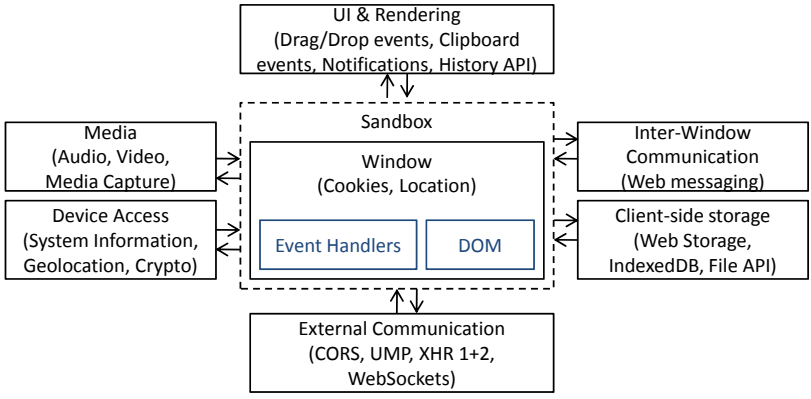


Figure 6.2: Synthesized model of the emerging HTML5 APIs.

Central in the model is the *window* concept, containing the document. The window manifest itself as a browser window, a tab, a popup or a frame, and provides access to the location and history, event handlers, the document and its associated DOM tree. Event handlers allow to register for a specific event (e.g. being notified of mouse clicks), and access to the DOM enables a script to read or modify the document's structure on the fly. Additionally, a *sandbox* can impose coarse-grained restrictions on an iframe, as mentioned in Section 6.2.2.

Inter-frame communication allows sending messages between windows (e.g. between mashup components). This includes window navigation, as well as Web Messaging (postMessage).

Client-side storage enables applications to temporarily or persistently store data. This can be achieved via Web Storage, IndexedDB or the File API.

External communication features such as CORS, UMP, XMLHttpRequest level 1 and 2, and websockets allow an application to communicate with remote websites, even in cross-origin settings.

Device access allows the web application to retrieve contextual data (e.g. geolocation) as well as system information such as battery level, CPU information and ambient sensors.

Media features enable a web application to play audio and video fragments, as well as capture audio and video via a microphone or webcam.

The UI and rendering features allow subscription to clipboard and drag-and-drop events, issuing desktop notifications and populating the history via the History API.

For a more thorough analysis of the HTML5 APIs, we would like to refer to an extensive security analysis we have carried out, commissioned by the European Network and Information Security Agency (ENISA) [DRDPP11].

Secure composition policy

The policy layer associates the secure composition policy with a mashup component, and deploys the necessary security controls via the underlying layers. As composition granularity, we have chosen the iframe level; i.e. mashup components are each loaded in their separate iframe.

In particular, within WebJail the secure composition policy is expressed by the mashup integrator, and attached to a particular component via a newly-introduced *policy* attribute of the iframe element of the component to be loaded.

```
<iframe src="http://untrusted.com/compX/"
        policy="https://integrator.com/compX.policy"/>
```

We have grouped the identified security-sensitive operations in the HTML5 APIs in nine disjoint categories, based on their functionality: DOM access, Cookies, External communication, Inter-frame communication, Client-side storage, UI & Rendering, Media, Geolocation and Device access.

For a third-party component, each category can be fully disabled, fully enabled, or enabled only for a self-defined whitelist. The whitelists contain category-specific entries. For example, a whitelist for the category "DOM Access" contains the ids of the elements that might be read from or updated in the DOM. The nine security-sensitive categories are listed in Table 6.1, together with their underlying APIs, the amount of security-sensitive functions in each API, and their WebJail whitelist types.

The secure composition policy expresses the restrictions for each of the security-sensitive categories, and an example policy is shown below. Unspecified categories are disallowed by default, making the last line in the example policy obsolete.

```
{ "framecomm" : "yes",
  "extcomm" : [ "google.com", "youtube.com" ],
  "device" : "no" }
```

Table 6.1: Overview of the sensitive JavaScript operations from the HTML 5 APIs, divided in categories.

Categories and APIs (# op.)	Whitelist
DOM Access	ElemReadSet, ElemWriteSet
DOM Core (17)	
Cookies	KeyReadSet, KeyWriteSet
cookies (2)	
External Communication	DestinationDomainSet
XHR, CORS, UMP (4)	
WebSockets (5)	
Server-sent events (2)	
Inter-frame Communication	DestinationDomainSet
Web Messaging (3)	
Client-side Storage	KeyReadSet, KeyWriteSet
Web Storage (5)	
IndexedDB (16)	
File API (4)	
File API: Dir. and Syst. (11)	
File API: Writer (3)	
UI and Rendering	
History API (4)	
Drag/Drop events (3)	
Media	
Media Capture API (3)	
Geolocation	
Geolocation API (2)	
Device Access	SensorReadSet
System Information API (2)	
Total number of security-sensitive operations: 86	

It is important to note that WebJails or regular frames can be used inside WebJails. In such a case, the functionality in the inner frame is determined by the policies imposed on enclosing frames, in addition to its own policy (if it has one, as is the case with a WebJail frame). Allowing sensible cascading of policies implies that "deeper" policies can only make the total policy more strict. If this were not the case, a WebJail with a less strict policy could be used to "break out" of the WebJail restrictions.

The semantics of a policy entry for a specific category can be thought of as a set. Let $\mathcal{V}$ be the set of all possible values that can be listed in a whitelist. The "allow all" policy would then be represented by the set $\mathcal{V}$ itself, a whitelist

would be represented by a subset $w \subseteq \mathcal{V}$ and the "allow none" policy by the empty set ϕ. The relationship "x is at least as strict as y" can be represented as $x \subseteq y$. Using this notation, the combined policy p of 2 policies a and b is the intersection $p = a \cap b$, since $p \subseteq a$ and $p \subseteq b$.

After loading, parsing and combining all the policies applicable to the WebJail protected iframe, the policy is enforced via the underlying layers.

6.4.2 Advice construction layer

The task of the advice construction layer is to build advice functions based on the high-level policy received from the policy layer, and apply these advice functions on the low-level security-sensitive operations via deep aspect technology in the deep advice weaving layer.

To do so, the advice construction layer applies a 2-step refinement process. For each category of the secure composition policy, the set of relevant APIs is selected. Next for each API, the individual security-sensitive operations are processed. Consider for instance that a whitelist of type "KeyReadSet"[1] is specified for the client-side storage in the composition policy. This is first mapped to the various storage APIs in place (such as Web Storage and File API), and then advice is constructed for the security-sensitive operations in the API (e.g. for accessing the *localStorage* object).

The advice function decides, based on the policy, whether or not the associated API function will be called: if the policy for the API function is "allow all", or "allow some" and the whitelist matches, then the advice function allows the call. Otherwise, the call is blocked.

On successful completion of its job, the advice construction layer has advice functions for all the security-sensitive operations across the nine categories relevant for the specific policy. Next, the advices are applied on the original operations via the deep advice weaving layer.

6.4.3 Deep aspect weaving layer

The *(advice, operation)* pairs received from the advice construction layer are registered into the JavaScript engine as deep advice. The result of this weaving is that the original API function is replaced with the advice function, and that all accesspaths to the API function now go through the advice function.

[1] Such a whitelist contains a set of keys that may be read

The advice function itself is the only place where a reference to the original API function exists, allowing it to make use of the original functionality when desired.

6.5 Prototype implementation

To show the feasibility and test the effectiveness of WebJail, we implemented a prototype by modifying Mozilla Firefox 4.0b10pre.The modifications to the Mozilla code are localized and consist of ±800 lines of new code (±300 JavaScript, ±500 C++), spread over 3 main files. The prototype currently supports the security-sensitive categories external and inter-frame communication, client-side storage, UI and rendering (except for drag/drop events) and geolocation.

Each of the three layers of the implementation will be discussed now in more detail.

6.5.1 Policy layer

The processing of the secure composition policy via the *policy* attribute happens in the frame loader, which handles construction of and loading content into frames. The specified policy URL is registered as the policy URL for the frame to be loaded, and any content loaded into this frame will be subject to that WebJail policy, even if that content issues a refresh, submits a form or navigates to another URL.

When an iframe is enclosed in another iframe, and both specify a policy, the combinatory rules defined in Section 6.4 are applied on a per-category basis. To ease up parsing of a policy file, we have chosen to use the JavaScript Object Notation (JSON).

Once the combined policy for each category has been calculated, the list of APIs in that category is passed to the advice construction layer, along with the combined policy.

6.5.2 Advice construction layer

The advice construction layer builds advice functions for individual API functions. For each API, the advice construction layer knows what functions are essential to enforce the policy and builds a specific advice function that enforces it.

```
function makeAdvice(whitelist) {
  var myWhitelist = whitelist;

  return function(origf, obj, vp) {
    if(myWhitelist.ROindexOf(vp[0])>=0) {
      return origf.ROapply(obj, vp);
    } else {
      return false;
    }
  };
}

myAdvice = makeAdvice(['foo', 'bar']);
registerAdvice(myFunction, myAdvice);
disableAdviceRegistration();
```

Listing 6.1: Example advice function construction and weaving.

The advice function is a function that will be called instead of the real function. It will determine whether or not the real function will be called based on the policy and the arguments passed in the function call. Advice functions in WebJail are written in JavaScript and should expect 3 arguments: a function object that can be used to access the original function, the object on which the function was invoked (i.e. the `this` object) and a list with the arguments passed to the function.

The construction of a rather generic example advice function is shown in Listing 6.1. The listing shows a function `makeAdvice`, which returns an advice function as a closure containing the whitelist. Whenever the advice function is called for a function to which the first argument (`vp[0]`) is either 'foo' or 'bar', then the original function is executed. Otherwise, the advice function returns false.

Note that in the example, `ROindexOf` and `ROapply` are used. These functions were introduced to prevent prototype poisoning attacks against the WebJail infrastructure. They provide the same functionality as `indexOf` and `apply`, except that they have the `JSPROP_READONLY` and `JSPROP_PERMANENT` attributes set so they cannot be modified or deleted.

Next, each *(advice, operation)* pair is passed on to the deep aspect weaving layer to achieve the deep aspect weaving.

6.5.3 Deep aspect weaving layer

The deep aspect weaving layer makes sure that all codepaths to an advised function pass through its advice function. Although the code from WebJail is the first code to run in a WebJail iframe, we consider the scenario that there can be code or objects in place that already reference the function to be advised. It is necessary to maintain the existing references to a function, if they exist, so that advice weaving does not break code unintentionally.

The implementation of the deep aspect weaving layer is inspired by ConScript. To register deep advice, we introduce a new function called `registerAdvice`, which takes 2 arguments: the function to advise (also referred to as the 'original' function) and its advice function. Line 14 of Listing 6.1 illustrates the usage of the `registerAdvice` function.

In Spidermonkey, Mozilla's JavaScript engine, all JavaScript functions are represented by `JSFunction` objects. A `JSFunction` object can represent both a native function, as well as a JIT compiled JavaScript function. Because WebJail enforces policies on JavaScript APIs and all of these are implemented with native functions, our implementation only considers `JSFunction` objects which point to native code[2].

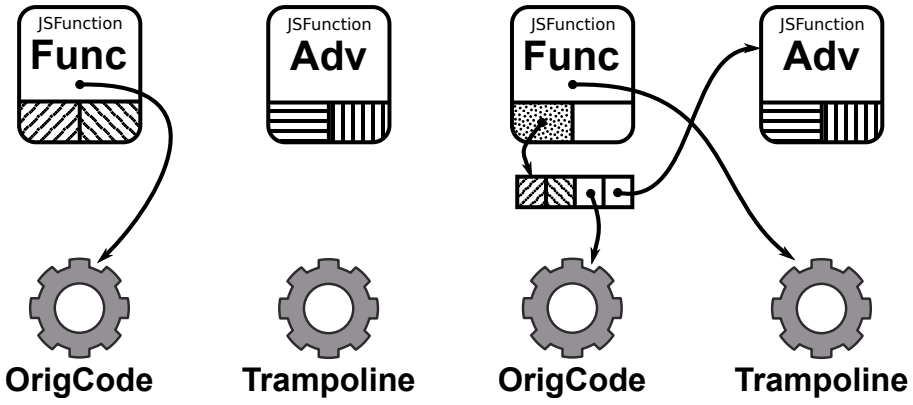
The process of registering advice for a function is schematically illustrated in Figure 6.3. Consider a native function `Func` and its advice function `Adv`. Before deep aspect weaving, the `JSFunction` object of `Func` contains a reference to a native C++ function `OrigCode`.

At weaving time, the value of the function pointer in `Func` (which points to `OrigCode`) and a reference to `Adv` are backed up inside the `Func` object. The function pointer inside `Func` is then directed towards the `Trampoline` function, which is an internal native C++ function provided by WebJail.

At function invocation time, the `Trampoline` function will be called as if it were the original function (`OrigCode`). This function can retrieve the values backed up in the weaving phase. From the backed up function pointer pointing to `OrigCode`, a new anonymous `JSFunction` object is created. This anonymous function, together with the current `this` object and the arguments to the `Trampoline` function are passed to the advice function `Adv`. Finally, the result from the advice function is returned to the calling code.

In reality, the `registerAdvice` function is slightly more complicated. In each `JSFunction` object, SpiderMonkey allocates 2 private values, known as "reserved slots," which can be used by Firefox to store opaque data. As shown in Figure 6.3,

[2]Although WebJail could be implemented for non-native functions as well.



(a) Before weaving (b) After weaving

Figure 6.3: Schematic view of deep aspect weaving.

the reserved slots of `Func` (hatched diagonally) are backed up in the weaving phase together with the other values. During invocation time, these reserved slots are then restored into the anonymous function mentioned earlier.

Note that all code that referenced `Func` still works, although calls to this function will now pass through the advice function `Adv` first. Also note that no reference to the original code `OrigCode` is available. The only way to call this code is by making use of the advice function.

To prevent any other JavaScript code from having access to the `registerAdvice` function, it is disabled after all advice from the policy has been applied. For this purpose, WebJail provides the `disableAdviceRegistration` function, which disables the use of the `registerAdvice` function in the current JavaScript context.

6.6 Evaluation

6.6.1 Performance

We performed micro-benchmarks on WebJail to evaluate its performance overhead with regard to page load-time and function execution. The prototype implementation is built on Mozilla Firefox 4.0b10pre, and compiled with the GNU C++ compiler v4.4.4-14ubuntu5. The benchmarks were performed on

an Apple MacBook Pro 4.1, with an Intel Core 2 Duo T8300 CPU running at 2.40GHz and 4GB of memory, running Ubuntu 10.10 with Linux kernel version `2.6.35-28-generic`.

Page load-time overhead

To measure the page load-time overhead, we created a local webpage (`main.html`) that embeds another local page (`inner.html`) in an iframe with and without a local policy file. `inner.html` records a timestamp (`new Date().getTime()`)) when the page starts and stops loading (using the body `onload` event). WebJail was modified to record the starttime before anything else executes, so that policy retrieval, loading and application is taken into account. After the results are submitted, `main.html` reloads.

We averaged the results of 1000 page reloads. Without WebJail, the average load-time was 16.22 ms ($\sigma = 3.74$ms). With WebJail, the average is 23.11 ms ($\sigma = 2.76$ms).

Function execution overhead

Similarly, we used 2 local pages (`main.html` and `inner.html`) to measure function execution overhead. `inner.html` measures how long it takes for 10000 iterations of a piece of code to execute. We measured 2 scenarios: a typical XMLHttpRequest invocation (constructor, `open` and `send` functions) and a localStorage set and get (`setItem` and `getItem`). Besides measuring a baseline without WebJail policy, we measured each scenario when restricted by 3 different policies: "allow all", "allow none" and a whitelist with 5 values. The averages are summarized in Table 6.2.

Table 6.2: Function execution overhead.

	XMLHttpRequest	localStorage
Baseline	1.25 ms	0.37 ms
"Allow all"	1.25 ms (+ 0%)	0.37 ms (+ 0%)
"Allow none"	0.07 ms (- 94.4%)	0.04 ms (- 89.2 %)
Whitelist	1.33 ms (+ 6.4%)	0.47 ms (+ 27%)

To conclude, we have registered a negligible performance penalty for our WebJail prototype: a page load-time of 7 ms, and an execution overhead in case of sensitive operations about 0.1 ms.

6.6.2 Security

As discussed in Section 6.5.3, the `registerAdvice` function disconnects an available function and makes it available only to the advice function. Because of the use of deep aspects, we can ensure that no other references to the original function are available in the JavaScript environment, even if such references already existed before `registerAdvice` was called. We have successfully verified this full mediation of the deep aspects using our prototype implementation.

Because advice functions are written in JavaScript and the advice function has the only reference to the original function, it would be tempting for an attacker to attack the WebJail infrastructure. The retrieval and application of a WebJail policy happens before any other code is executed in the JavaScript context. In addition, the `registerAdvice` function is disabled once the policy has been applied. The only remaining attack surface is the advice function during its execution. The advice functions constructed by the advice construction layer are functionally equivalent to the example advice function created in Listing 6.1. We know of 3 attack vectors: prototype poisoning of `Array.prototype.indexOf` and `Function.prototype.apply`, and `toString` redefinition on `vp[0]` (the first argument to the example advice function in Listing 6.1). By introducing the readonly copies `ROindexOf` and `ROapply` (See Section 6.5.2), we prevent an attacker from exploiting the first 2 attack vectors. The third vector, `toString` redefinition, was verified in our prototype implementation and is not an issue because `toString` is never called on the argument `vp[0]`.

6.6.3 Applicability

To test the applicability of the WebJail architecture, we have applied our prototype implementation to mainstream mashup platforms, including iGoogle and Facebook. As part of the setup, we have instrumented responses from these platforms to include secure composition policies, by automatically injecting a *policy* attribute in selected iframes. Next, we have applied both permissive composition policies as well as restricted composition policies and verified that security-sensitive operations for the third-party components were executed as usual in the first case, and blocked in the latter case. For instance, as part of the applicability tests, we applied WebJail to control Geolocation functionality in the Google Latitude [Goob] component integrated into iGoogle, as well as external communication functionality of the third-party Facebook application "Tweets To Pages" [Inv] integrated into our Facebook page.

6.7 Discussion and future work

In the previous sections, we have showed the feasibility of the WebJail architecture via a prototype implementation in Firefox, and evaluated the performance, security and applicability. By applying micro-benchmarks, we measured a negligible overhead, we discussed how the WebJail architecture achieves full mediation via deep aspect weaving, and we briefly illustrated the applicability of WebJail in mainstream mashup platforms.

In this section, we will discuss some points of attention in realizing least-privilege integration in web mashups and some opportunities for further improvements.

First, the granularity chosen for the secure composition policies for WebJail is primarily driven by the ease of configuration for the mashup integrator. We strongly believe that the category level of granularity increases the adoption potential by integrators and browsers, for instance compared to semantically rich and expressive security policies as is currently the case in wrapper approaches or ConScript. In fact, we chose to introduce this policy abstraction to let the integrator focus on the "what" rather than the "how." A next step could be to define policy templates per mashup component type (e.g. advertisement and geotagging components).

Nevertheless, more fine-grained policies could also be applied to achieve least-privilege integration, but one should be aware of the potential risk of creating an inverse sandbox. The goal of a least-privilege integration architecture, such as WebJail, is to limit the functionality available to a (possibly) malicious component. In case the policy language is too expressive, an attacker could use this technology to achieve the inverse. An attacker could integrate a legitimate component into his website and impose a malicious policy on it. The result is effectively a hardcoded XSS attack in the browser. For instance, the attacker could introduce an advice that leaks all sensitive information out of a legitimate component as part of its least-privilege composition policy without being stopped by the Same-Origin Policy.

One particular area where we see opportunities for more fine-grained enforcement are cross-domain interactions. Ongoing research on Cross-Site Request Forgery (CSRF) [DRDH+10, DRDJP11, Sam11, Mao11] already differentiates between benign and potentially malicious cross-domain requests, and restricts the latter class as part of a browser extension. This line of research could be seen as complementary to the presented approach, and a combination of both would allow a more fine-grained enforcement for cross-domain interactions.

Second, a possible technique to escape a modified JavaScript execution context in an iframe, would be to open a new window and execute JavaScript in there.

We have anticipated this attack by hardcoding policies for e.g. the `window.open` function. This is however not the best approach. The upcoming HTML 5 specs include the sandbox attribute for iframes. This specification states that a sandbox should prevent content from creating new auxiliary browsing contexts. Mozilla Firefox does not support the sandbox attribute yet. The hardcoded policy for `window.open` is a quick fix while we are working on our own full implementation of the sandbox attribute in Mozilla Firefox.

Another way to escape WebJail is to access the window object of the parent or a sibling frame and make use of the functions in that JavaScript context (e.g. `parent.navigator.geolocation.getCurrentPosition`). In such a scenario, accessing another JavaScript context falls under the Same-Origin Policy and will only be possible if both the caller and callee are in the same origin. To avoid this attack, the WebJail implementation must restrict access to sensitive operations in other execution contexts under the Same-Origin Policy.

Thirdly, the categories in the policy files of WebJail are a result of a study of the sensitive JavaScript operations in the new HTML5 APIs. Most of the HTML5 APIs are working drafts and might change in the future. The category list in WebJail is therefore an up-to-date snapshot, but might be subject to change in the future. Even after the specifications for HTML5 are officially released, the functionality in browsers might keep changing. To cope with this evolving landscape, WebJail can easily be extended to support additional categories and APIs as well.

Finally, the WebJail architecture is tailored to support least-privilege integration in mashups that are built via iframe-integration. An interesting future track is to investigate how to enable browsers to support least-privilege script-inclusion integration as well. Since in such a scenario, one cannot build on the fact that a separate execution context is created, we expect this to be a challenging trajectory.

6.8 Related Work

There is a broad set of related work that focuses on the integration of untrusted JavaScript code in web applications.

JavaScript subsets. A common technique to prevent undesired behavior is to restrict the untrusted code (i.e. the third-party component) to a safe subset of JavaScript. The allowed operations within the subset prevent the untrusted code from obtaining elevated privileges, unless explicitly allowed by the integrator.

ADSafe [Cro] and FBJS [The] requires third-party components to be written in a JavaScript subset that is known to be safe. The ADSafe subset removes several unsafe features from JavaScript (e.g. global variables, eval, ...) and provides safe alternatives through the `ADSAFE` object. Caja [MSL+08], Jacaranda [Jac] and Live Labs' Websandbox [Micj] take a different approach. Instead of heavily restricting the developer's language, they transform the JavaScript code into a safe version. The transformation process is based on both static analysis and rewriting to integrate runtime checks.

These techniques effectively support client-side least-privilege integration of mashup components. The main disadvantage is the tight coupling of the security features with the third-party component code. This requires control over the code, either at development or deployment time, which conflicts with legacy components and remote component delivery (R2), and reduces the applicability to mashup scenarios where the integrator delivers the components to the browser.

JavaScript instrumentation and access mediation. Instead of restricting a third-party component to a JavaScript subset, access to specific security-sensitive operations can be mediated. Mediation can consist of blocking the call, or letting a policy decide whether or not to allow it.

BrowserShield [RDW+06] is a server-side rewriting technique, that rewrites certain JavaScript functions to use safe equivalents. These safe equivalents are implemented in the "bshield" object that is introduced through the BrowserShield JavaScript libraries that are injected into each page. BrowserShield makes use of a proxy to inject its code into a webpage.

Self-protecting JavaScript [PSC09, MPS10] is a client-side wrapping technique that applies advice around JavaScript functions, without requiring any browser modifications. The wrapping code and advice are provided by the server and are executed first, ensuring a clean environment to start from. The advice is *non-deep* advice, meaning that by protecting one operation, different access paths to the same operation are not automatically protected. The main challenge of this approach is to ensure full mediation (R1) without breaking the component's legitimate functionality (e.g. via removal of prototypes), since both policy and third-party component code live in the same JavaScript context.

Browser-Enforced Embedded Policies (BEEP) [JSH07] injects a policy script at the server-side. The browser will call this policy script before loading another script, giving the policy the opportunity to vet the script about to be loaded. The loading process will only continue after the approval of the policy. This approach offers control over which scripts are loaded, but is too coarse grained to assign privileges to specific components.

ConScript [ML10] allows the enforcement of fine-grained security policies for JavaScript in the browser. The approach is similar to self-protecting JavaScript, except that ConScript uses *deep* advice, thus protects all access paths to a function. The price for using deep advice is the need for client-side support in the JavaScript engine. A limitation of ConScript is that policies are not composition policies: the policies are provided by and applied to the same webpage, which conflicts with remote component delivery (R2) and secure composition policy configurable by the integrator (R3).

In contrast to the techniques described above, WebJail offers the integrator the possibility to define a policy that restricts the behavior of a third-party component in an isolated way. Additionally, all of the techniques above use JavaScript as a policy language. This amount of freedom complicates the writing of secure policies: protection against all the emerging HTML5 APIs is fully up to policy writer and can be error-prone, a problem that the WebJail policy language is not susceptible to.

Web application code and data analysis. A common protection technique against XSS vulnerabilities or attacks is server-side code or data analysis. Even though these techniques can only be used to check if a component matches certain security requirements and do not enforce a policy, we still discuss them here, since they are a server-side way to ensure that a component meets certain least-privilege integration requirements *out-of-the-box*.

Gatekeeper [GL09] is a mostly static [sic] enforcement mechanism designed to defend against possibly malicious JavaScript widgets on a hosting page. Gatekeeper analyzes the complete JavaScript code together with the hosting page. In addition, Gatekeeper uses runtime enforcement to disable dynamic JavaScript features.

XSS-Guard [BV08] aims to detect and remove scripts that are not intended to be present in a web application's output, thus effectively mitigating XSS attacks. XSS-Guard dynamically learns what set of scripts is used for an HTTP request. Using this knowledge, subsequent requests can be protected.

Recently, Mozilla proposed the Content Security Policy (CSP) [SSM10], which allows the integrator to insert a security policy via response headers or meta tags. Unfortunately, CSP only supports restrictions on a subset of the security-sensitive operations discussed in this chapter, namely operations potentially leading to content injection (e.g. script inclusion and XHR).

Information flow control. Information flow control techniques can be used to detect unauthorized information sharing or leaking between origins or external

parties. This is extremely useful for applications that are allowed to use sensitive data, such as a location, but are not allowed to share that data.

Both Magazinius et al. [MAS10] and Li et al. [LZW10] have proposed an information flow control technique that prevents unauthorized sharing of data. Additionally, both techniques support authorized sharing by means of declassification, where a certain piece of data is no longer considered sensitive.

Secure multi-execution [DP10] detects information leakage by simultaneously running the code for each security level. This approach is a robust way to detect information leakage, but does not support declassification.

Information flow control techniques themselves are not suited for enforcing least-privilege integration. Likewise, WebJail is not suited to enforce information flow control, since it would be difficult to cover all possible leaks. Both techniques are complementary and can be used together to ensure least-privilege integration without unauthorized information leaking.

Isolating content using specialized HTML. Another approach to least-privilege integration is the isolation of untrusted content. By explicitly separating the untrusted code, it becomes easier to restrict its behavior, for example by preventing script execution.

The "untrusted" attribute [FHEW08] on a div element aims to allow the browser to make the difference between trusted and untrusted code. The idea is to enclose any untrusted content with such a div construct. This technique fails to defend against injecting closing tags, which would trivially circumvent the countermeasure.

The new "sandbox" attribute of the iframe element in HTML 5 [HH10] provides a safer alternative, but is very coarse-grained. It only supports limited restrictions, and as far as JavaScript APIs are concerned, it only supports to completely enable or disable JavaScript.

ADJail [LGV10] is geared towards securely isolating ads from a hosting page for confidentiality and integrity purposes, while maintaining usability. The ad is loaded on a shadow page that contains only those elements of the hosting page that the web developer wishes the ad to have access to. Changes to the shadow page are replicated to the hosting page if those changes conform to the specified policy. Likewise, user actions on the hosting page are mimicked to the shadow page if allowed by the policy. ADJail limits DOM access and UI interaction with the component, but does not restrict the use of all other sensitive operations like WebJail can.

User-provided policies. Mozilla offers *Configurable Security Policies* [Rud], a user-configurable policy that is part of the browser. The policy allows the user to explicitly enable or disable certain capabilities for specific internet sites. An example is the option to disallow a certain site to open a popup window. Some parts of this idea have also been implemented in the Security zones of Internet Explorer.

The policies and enforcement mechanism offered by this technique resemble WebJail. The major difference is that these policies are user-configurable, and thus not under control of the integrator. Additionally, the policies do not support a different set of rules for the same included content, in two different scenarios, whereas WebJail does.

6.9 Conclusion

In this chapter we have presented WebJail, a novel client-side security architecture to enables least-privilege integration of third-party components in web mashups. The WebJail security architecture is compatible with legacy mashup components, and allows the direct delivery of components from the service providers to the browser.

We have designed a secure composition language for WebJail, based on a study of security-sensitive operations in HTML5 APIs, and achieved full mediation by applying deep aspect weaving within the browser.

We have implemented a prototype of WebJail in Mozilla Firefox 4.0, and applied it successfully to mainstream platforms such as iGoogle and Facebook. In addition, we have evaluated the performance of the WebJail implementation using micro-benchmarks, showing that both the page load-time overhead ($\pm$7ms) and the execution overhead of a function advised with a whitelist policy ($\pm$0.1ms) are negligible.

Acknowledgments

This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, IBBT, IWT, the Research Fund K.U.Leuven and the EU-funded FP7-projects WebSand and NESSoS.

The authors would also like to thank Maarten Decat and Willem De Groef for their contribution to early proof-of-concept implementations [Dec10, Wil] to test the feasibility of the presented research.

Chapter 7

JSand: Complete Client-Side Sandboxing of Third-Party JavaScript without Browser Modifications

Publication data

Contained in this chapter is the paper titled "JSand: Complete Client-Side Sandboxing of Third-Party JavaScript Without Browser Modifications," as presented at the 2012 Annual Computer Security Applications Conference (ACSAC 2012) [AVAB+12]. The main authors of this work are Pieter Agten, Steven Van Acker and Yoran Brondsema. Steven Van Acker was mainly responsible for the inception of the idea and the evaluation of the prototype.

Preamble

This chapter looks into JavaScript sandboxing without browser modifications.

This chapter describes JSand, a JavaScript sandboxing system that does not require any browser modifications.

JSand relies on an object-capability environment to sandbox untrusted JavaScript. In such an object-capability environment, all security-sensitive JavaScript functionality can be encapsulated in objects. These objects can only be accessed through unforgeable references passed to JavaScript code that requires the security-sensitive functionality encapsulated in those objects.

Google's Caja team identified a subset of JavaScript of which they proved that it maintains an object-capability environment, called the Secure ECMAScript (SES) subset. Furthermore, they developed a JavaScript library which allows the execution of Secure ECMAScript on ECMAScript 5 compatible browsers.

JSand relies on this SES library to create an object-capability environment in which untrusted JavaScript can be executed. Using JSand, untrusted code is passed a reference to a version of the DOM wrapped with a membrane. This membrane implements the Membrane pattern by using ECMAScript 6's Proxy API, thus ensuring that all references to other objects retrieved by calling methods in this wrapped DOM, are also wrapped by the membrane.

Because all access to the DOM are now mediated by the membrane in which the DOM is wrapped, JSand can enforce any access policy on JavaScript running inside a JSand sandbox.

JSand has support for dynamically loaded scripts and is backwards compatible with legacy JavaScript that does not comply with the SES subset, by rewriting it in a support layer.

We have implemented JSand and evaluated it on the Google Chrome browser, showing acceptable overhead. Furthermore, JSand was evaluated using a sandboxed version of Google Maps, Google Analytics and an application built on jQuery. All applications ran without problems in their JSand sandbox. We performed performance measurements on the Google Maps application to determine JSand's overhead. The load time of the Google Maps application rose from 308 ms when running natively, to 1433 ms when running in the sandbox. This overhead is mostly due to the rewriting of legacy JavaScript in the support layer. Once loaded, a user experiences a 31% delay when interacting with the application, which we believe to be quite reasonable.

The main contributions of this research are:

- JSand is the first JavaScript sandbox that does not need browser modifications, while completely mediating access to the DOM,
- JSand is secure, compatible with real world web applications such as Google Maps, Google Analytics and jQuery, and performs reasonably well.

In hindsight, JSand was the first JavaScript sandboxing mechanism that used a SES environment in combination with a wrapped global object and did not require any browser modifications. JSand is completely built on top of new JavaScript 5 APIs such as JavaScript strict mode and the Proxy API. In addition, it was compatible with legacy scripts like the Google Maps API by rewriting them in the browser when needed.

At the time, the JSand implementation was no more than a prototype capable of running the web applications required for its evaluation. Since JSand was published, its code has been refined with extra functionality that allows the virtual DOM inside the membrane wrapper to be easily debugged and augmented with missing properties. New demos showcase fully functional jQuery user interfaces and even a version of the Tetris game. With the added debugging support, it is now trivial to detect missing properties and add them to the virtual DOM. Furthermore, JSand was extended so that multiple sandboxes in a single JavaScript context could easily be created without interfering with each other. In addition, JSand was adapted to enforce user-friendly policies such as WebJail and inserted into a Java JEE framework that automatically outputs the boilerplate HTML code necessary to integrate JSand into a web application.

JSand shows that browser modifications are not needed to achieve client-side JavaScript sandboxing by making use of browser functionality that exists today. In the future we can most likely expect new JavaScript functionality that can be used to further optimize and enhance JSand, or any other JavaScript sandboxing system that equally does not require browser modifications.

Abstract

The inclusion of third-party scripts in web pages is common practice. A recent study has shown that more than half of the Alexa top 10 000 sites include scripts from more than 5 different origins. However, such script inclusions carry risks, as the included scripts operate with the privileges of the including website.

We propose JSand, a server-driven but client-side JavaScript sandboxing framework. JSand requires *no browser modifications*: the sandboxing framework is implemented in JavaScript and is delivered to the browser by the websites that use it. Enforcement is done *entirely at the client side*: JSand enforces a server-specified policy on included scripts without requiring server-side filtering or rewriting of scripts. Most importantly, JSand is *complete*: access to all resources is mediated by the sandbox.

We describe the design and implementation of JSand, and we show that it is secure, backwards compatible, and that it performs sufficiently well.

7.1 Introduction

In the last decade, the web platform has become the number one platform on the Internet. There is a clear paradigm shift from desktop applications and proprietary client-server solutions towards web-enabled services. An important catalyst for this paradigm shift has been the power of JavaScript as well as the advent of HTML5, giving web developers the tools to build rich and interactive websites.

As a consequence of this enormous growth in popularity, the web has also become the primary attack platform: SANS [SAN09] reported in 2009 that more than 60% of all cyber attacks are aimed at web applications, and more than 80% of discovered vulnerabilities are web-related. A whole range of web attacks exists in the wild, ranging from Cross-Site Scripting, Cross-Site Request Forgery and SQL injection to the exploitation of broken authorization and session management. This chapter focuses on one particular and important class of web attacks, namely attacks due to the insecure integration of JavaScript.

To enrich the functionality and interaction of a website, a common and widespread approach is to integrate JavaScript from third-party script providers. Recent studies [YW09, NIK+12] have shown that 96.9% of websites include scripts from external sources, and on average each website includes scripts from 3.1 external sources. For example, websites integrate among others JavaScript-enabled advertisements (such as Google AdSense and adBrite), Web analytics

frameworks (such as Google Analytics, Yahoo! Web Analytics and Tynt), web widgets and buttons (such as Google Maps, addToAny button and Google +1 button), and JavaScript programming libraries (such as jQuery and Dojo). The popular news site nytimes.com for instance, includes 17 pieces of third-party JavaScript code, hosted on 6 unique domains.

The de facto browser security model today is defined by the Same-Origin Policy (SOP). The SOP restricts access of client-side scripts to resources belonging to the same origin[1]. For instance, the SOP ensures that document data and cookies from one origin cannot be read by scripts belonging to another origin. However, the SOP includes some important relaxations with respect to navigation and content inclusion (e.g. embedded images, scripts) [Zal10]. In particular, if a page from one origin includes a script from another origin, the included script is treated as if it belongs to the including origin, and hence it inherits all the capabilities and permissions of the hosting page. This makes malicious script inclusion a very powerful attack vector.

Several countermeasures have been proposed to limit the capabilities of third-party JavaScript, including (1) the introduction of safe subsets of JavaScript [The, Cro, MT09], (2) client-side reference monitors [ML10, ARD+11], and (3) server-side transformations of the script to be included [MSL+08, LGV10]. However, all of these have at least one of the following limitations.

First, some approaches [ML10, ARD+11] require intrusive *browser modifications*, in particular to the JavaScript engine and the binding between browser and JavaScript engine. Such intrusive browser modifications hinder short-term deployment of the countermeasure.

Second, some approaches do *not support client-side script inclusion*: in order to perform server-side pre-processing (e.g. source-to-source translation or filtering) of the scripts, the scripts have to pass through the web server [MSL+08, The, Cro]. This effectively changes the architectural model of client-side script inclusion to server-side script inclusion.

Third, some approaches do *not provide complete mediation* between different scripts on the same page, or to all resources exposed in the browser. Self-Protecting JavaScript (SPJS) [PSC09, MPS10] assumes that all scripts included on a hosting page need identical security constraints. It does not differentiate between different external scripts nor between local and remote inclusions. AdJail [LGV10] successfully isolates untrusted advertisements from the Document Object Model (DOM) of the hosting page, but since it uses

[1] An origin is a (protocol, domain name, port) tuple.

iframes as isolation units, it cannot fully protect security-sensitive APIs such as XHR, Geolocation and local storage.

Inspired by recent advances in achieving object-capability guarantees for JavaScript [MSL+08, MMT10, Mil, Hei11], this chapter presents JSand, a novel security architecture to securely integrate third-party JavaScript. We improve upon the state-of-the-art with the following contributions:

1. JSand is the first JavaScript sandbox that (1) does not need browser modifications, (2) supports client-side script inclusion and (3) completely mediates different scripts and the browser APIs.
2. We show evidence that JSand is secure, compatible with complex and widely used scripts (such as Google Maps, Google Analytics and jQuery) and performs sufficiently well.

The rest of this chapter is structured as follows. Section 7.2 introduces the necessary background and defines the problem statement. In Section 7.3, the JSand architecture is presented, and Section 7.4 discusses several relevant implementation aspects. Section 7.5 evaluates the security, compatibility and performance of JSand. Finally, Section 7.6 discusses related work, and we conclude in Section 7.7.

7.2 Problem statement

7.2.1 Integrating third-party JavaScript

To enrich the functionality and interactivity of a website, a common and wide-spread approach is to integrate JavaScript from third-party script providers. The two most wide-spread techniques to integrate third-party JavaScript in web pages are through script inclusion and via iframe integration [RDD+10].

Script inclusion
HTML script tags are used to execute JavaScript as part of a web page. If the JavaScript code is integrated from an external source, the browser will still execute the code within the security context of the web page, without any restrictions of the SOP.

Iframe integration
HTML iframe tags allow a web developer to include one document inside another. The advantage of iframe integration is that the integrated

document is loaded in its own security context: integrated content from another origin is isolated from the integrating web page by the SOP.

Script inclusion is the de facto script integration technique on the web, both for local scripts as well as for external scripts. The iframe integration technique is used for web gadgets that do not have strong integration needs with the embedding web page, or have an out-of-band service-to-service communication channel (such as the Facebook Like button or Facebook Apps). In the remainder of this chapter, we focus on third-party JavaScript integration through script inclusion.

7.2.2 Malicious script inclusion

The browser security model for integrating third-party JavaScript is problematic. Once included in a website, a malicious script cannot only access all the document data and cookies, but with the advent of HTML5, the malicious script has also access to local storage data (e.g. Web Storage, IndexedDB), intra-window communication (Web Messaging), remote resource fetching via XHR and user-consented privileges (such as Geolocation, media capture, access to System Information API, and many more). This makes malicious script inclusion a very powerful attack vector. One can distinguish two types of attackers.

Malicious script provider
: The script provider has malicious intentions (but covers up by providing appealing functionality to potential customers), or becomes malicious over time (e.g. intentionally, or by selling out or quitting his business)

Benign script provider under attack
: The provider has no malicious intentions, but the scripts delivered to its clients become under control of an attacker. This can be due to the inclusion of other untrusted resources (e.g. in advertisement networks), due to a bug in the delivered script (e.g. a DOM-based XSS vulnerability [Kle05]), due to a server-side take-over (e.g. via SQL injection) or due to in-transit tampering with the scripts by a network attacker.

In both cases, the attacker controls the scripts included by the hosting page, and by default gains full access to the execution environment of the web page.

7.2.3 Requirements

Given the wide spread of script inclusion and the increasing impact of malicious script inclusion, there is a clear need for a novel security architecture to **securely integrate third-party JavaScript**, but **without introducing disruptive change**. Preserving backwards compatibility is crucial in the web context. We therefore identify the following requirements:

R1 Complete mediation
All access to security-sensitive functionality should be completely mediated by the security mechanism. This includes access to the DOM, as well as security-sensitive JavaScript APIs (such as Geolocation and local storage). The attacker must be unable to circumvent the security mechanisms in place.

R2 Backwards compatible
The security mechanism should seamlessly operate in the current web ecosystem, i.e. it should not rely on browser modifications or disable the direct delivery of scripts from script provider to the browser. In addition, the security mechanism should support the integration of legacy scripts.

R3 Performance
The security mechanism should introduce only a minimal performance penalty, unnoticeable to the end-user.

7.3 JSAND security architecture

The JSand architecture enables the owner of a website to securely integrate third-party scripts, without needing disruptive change to either server-side or client-side infrastructure. We first give a high-level overview of the architecture and then discuss the architectural choices under the hood.

7.3.1 Architectural overview

Figure 7.1 depicts the JSand architecture. A website owner deploys JSand by including the JSand JavaScript library in his web pages. When one of these pages is loaded in a visitor's browser, the third-party scripts to be sandboxed are fetched directly from the servers of the script provider. The JSand library confines each script to its own secure sandbox, which isolates the script from other scripts and from the DOM.

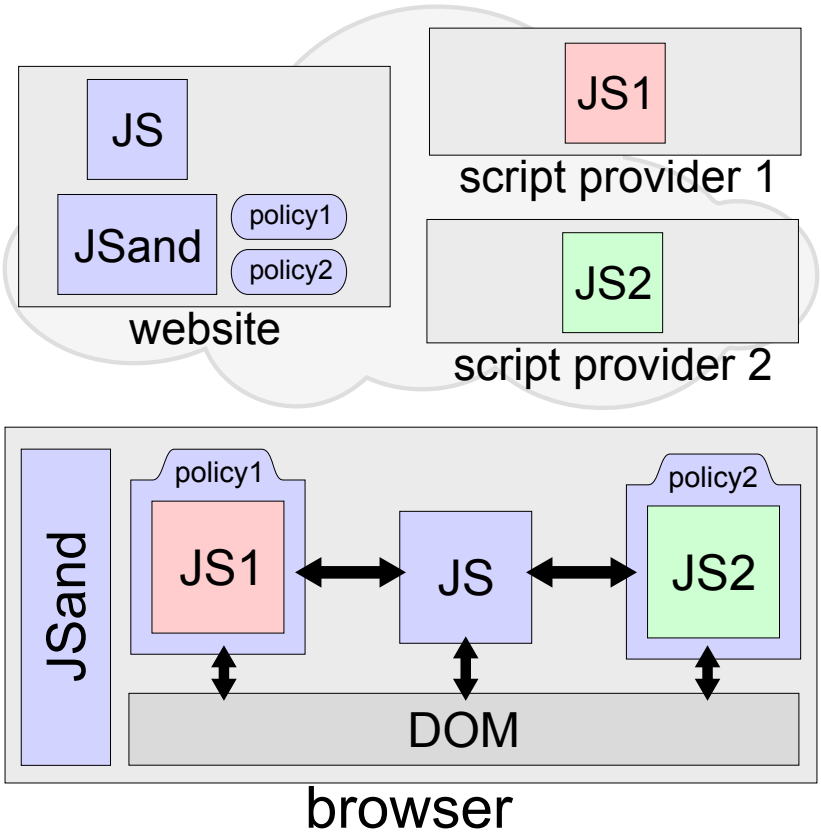


Figure 7.1: the JSand architecture. Inside the browser, all access from JSand sandboxes to the JavaScript environment is mediated according to server-supplied policies.

7.3.2 Under the hood

The JSand architecture is based upon the secure confinement of third-party JavaScript. JSand realizes this through the use of an object-capability environment. Such environment provides an appropriate device for isolating untrusted JavaScript: without an explicit and unforgeable reference to a security-sensitive object or function, a script is unable to access the resource or make use of its capabilities. The object-capability model is at the basis of Caja [MSL+08], and many other safe subsets of JavaScript [MMT10].

The JSand library invokes third-party JavaScripts with an initially minimal set of

capabilities (i.e. unforgeable references). To maintain control over all references acquired by a sandboxed script, JSand applies the Membrane pattern proposed by Miller [Mil06]. Our implementation of this pattern consists of placing policy-enforcing wrappers around objects that provide potentially security-sensitive operations. Whenever one of these objects returns a reference to another object, the membrane is extended to cover that object as well. This ensures a sandboxed script never has direct access to a security-sensitive operation.

The membrane's wrappers intercept all operations performed on the objects they wrap and hence implement the decision points for the security policy. On each decision point, the wrapper consults the security policy to determine whether or not the corresponding operation is permitted. If not, this will be indicated by the security policy and the operation will be blocked. The architecture is not bound to any specific type of security policy, which gives website owners the freedom to enforce arbitrarily complex policies.

Since all interactions between a script and the browser are performed by calling DOM methods, it suffices to place a wrapper around each DOM object in order to enforce a policy on all security-sensitive operations. These include not only operations to read or modify content of the hosting page, but also to communicate with other scripts and to use browser-provided JavaScript APIs.

In conclusion, the JSand architecture provides an end-to-end solution for securely integrating third-party JavaScripts on a website. The website owner is able to define and enforce security policies on third-party scripts, which puts him back into the driver's seat. JSand does not require disruptive change to the architecture of the web: it does not break direct script delivery towards the browser, and can be deployed without additional server-side or client-side infrastructure. The combination of the object-capability model and the Membrane pattern ensures that all access from a sandboxed script to security-sensitive operations passes through a membrane's wrappers, which enforce the security policy.

7.4 Prototype implementation

This section reports on the development of a mature JSand prototype, which is designed to work in ECMAScript 5 (ES5) compatible browsers with support for the proxy features of the upcoming ES Harmony standard. The current prototype runs seamlessly in Google Chrome v20.0.1132.21.

In Sections 7.4.1 and 7.4.2, we present the client-side technology for executing third-party JavaScript in a confined sandbox. Section 7.4.3 describes the type

of security policies that are enforced. Next, Section 7.4.4 illustrates how access to security-sensitive operations is completely mediated. Section 7.4.5 discusses how our prototype deals with dynamic script loading and Section 7.4.6 describes a set of automatic script transformations to improve compatibility with legacy scripts.

7.4.1 Object-capability system

As described in Section 7.3.2, the JSand architecture relies on an object-capability environment to provide complete mediation. The ECMAScript language does not qualify as an object-capability language by itself. For instance, any script has access to all global variables by default, and consequently has capabilities that are not under control of any security framework. However, in 2008 Miller et al. [MSL+08] have identified a subset of ES3 which forms a true object-capability language. More recently, the Google Caja team has identified a subset of ES5 strict, named Secure ECMAScript (SES), that also provides such an object-capability language. Moreover, they have developed a JavaScript library that enables the execution of SES on ES5-compatible browsers [Mil]. This library provides methods for safely evaluating SES-compliant code in an isolated environment. A key feature of the library is that it can execute completely at the client side and hence does not rely on any custom server-side architecture. JSand uses the SES library to realize its underlying object-capability environment.

However, since SES is a subset of ES5 strict, which in turn is a subset of ES5 non-strict, not all currently deployed JavaScripts are SES-compliant. Furthermore, the language supported by the SES library differs from true SES in several minor ways, further reducing compatibility with legacy scripts. Two important incompatibilities between ES5 and the SES-like language supported by the SES library are described below.

Global variables
: In ES5, the global `window` object can have arbitrary properties and for each of these properties there is a corresponding global variable with the same name. Conversely, for any global variable, a corresponding property with the same name is defined on the global object. In SES, this is no longer the case: global variables are not aliased by properties on the global object or vice versa.

Strict mode
: SES enforces strict mode for all scripts. Hence, ES5 non-strict code might be incompatible with SES. For instance, strict mode drops support for the `with` keyword, prevents the introduction of new variables into the

outer scope by an `eval` and no longer binds `this` to the global object in a function call.

SES was designed to support (only) recognized ES5 best practices. Therefore, scripts that adhere to these best practice standards are SES-compliant and hence we expect the number of fully SES-compliant scripts to increase progressively as these best practices become more widespread. Although not all legacy script run without errors under the SES library, the secure confinement of these scripts is never at stake. Nevertheless, we have developed a support layer to improve compatibility with legacy scripts. This layer is described in detail in Section 7.4.6.

To enforce the object-capability model and to provide support for legacy scripts, the SES library and the support layer need access to the source code of scripts to be sandboxed. Our prototype fetches this code using the XMLHttpRequest (XHR) API. By default, this API is subject to the SOP, but recently added web features have facilitated cross-domain interactions, namely Cross-Origin Resource Sharing (CORS) [W3Ch] and the Uniform Messaging Policy (UMP) [W3Cj]. In case CORS or UMP are not supported by the script provider, our solution can fall back to a server-side JavaScript proxy [Yah].

7.4.2 Policy-enforcing membranes

The Proxy API

For implementing the Membrane pattern in an efficient and transparent way, our prototype uses the Harmony Proxy API, which is scheduled to be standardized in the next version of ECMAScript [CM10]. This API enables us to create wrappers that generically intercept all property accesses and assignments on specific objects. The following code example depicts how this works.

```
function wrap(target, policy) {
  var handler = {
    get: function(proxy, propertyName) {
      if (policy.isGetAllowed(propertyName) {
        return target[name];
      }
      return null;
    }
    set: function(proxy, propertyName, value) {
      if (policy.isSetAllowed(propertyName) {
        target[name] = value;
        return true;
      }
      return false;
    }
  }
  return Proxy.create(handler, Object.getPrototypeOf(target));
}
```

The `wrap` function creates a simple policy-enforcing wrapper around a specific `target` object. All property accesses and assignments on this wrapper are intercepted by the `get` and `set` traps of the handler object, which uses the `policy` object to determine whether or not the access or assignment is allowed.

Membrane implementation

To implement the Membrane pattern, the handlers used in JSand transitively wrap all objects they return from the `get` trap and unwrap the objects they receive in the `set` trap. The entire prototype chain of a wrapper must be wrapped as well, to prevent an attacker from piercing the membrane by accessing an unwrapped prototype.

If an object to be returned from the `get` trap is a function, a function proxy that wraps the original function is returned. This function proxy first unwraps all its arguments, then calls the original function using these unwrapped arguments and finally wraps the return value before returning it to the caller, thereby further expanding the metaphorical membrane. Some methods, such as `window.addEventListener`, take a callback function as an argument; like all other arguments, this callback must be wrapped appropriately to uphold the membrane. Because a callback function is executed in the context of a sandbox, its wrapper must wrap each of its arguments and must unwrap the return value after calling the original function with the wrapped arguments.

Each sandbox keeps a mapping from each of its wrappers to the target object it wraps and vice versa. This makes it possible to unwrap previously wrapped objects and to ensure that there is at most one wrapper (per sandbox) corresponding to each target object, making the membrane identity-

preserving [VCM12]. The mapping from wrappers to their corresponding targets is only accessible from outside the sandbox, for otherwise an attacker could use it to escape from the sandboxed environment.

Whereas sandboxed code should always be confined to the bounds of its own sandbox, many use cases require an operation to introduce code from outside a sandbox into an existing sandbox. Such operations enable a website owner to extend or interact with a sandboxed script. JSand sandboxes provide two functions for introducing new code into them: `innerEval(code)` and `innerLoadScript(url)`. The first function evaluates a literal code string, while the second loads a script at a given URL.

In conclusion, the Membrane pattern transparently isolates a sandbox from code running outside of it or in other sandboxes. Since the handlers intercept each property access and assignment made on a wrapper, they contain the enforcement points which consult the security policy to determine whether or not an operation is permitted.

7.4.3 Security policies

Defining good security policies is important for ensuring the secure confinement of sandboxed scripts. To avoid needing a *known-good* version of a script to be sandboxed, a policy should be based on the claimed functionality of a script, as opposed to being based on actions performed by any specific version of the script. Generic templates can be provided to support website owners in defining good security policies.

Since the JSand architecture is independent of the specific type of security policy to enforce, policies can range from simple stateless policies, to arbitrarily complex policies. In both cases, the security policy can be specified as a JavaScript function that takes information about the operation to be performed as input and returns a boolean indicating whether or not the operation is allowed. We discuss three types of policies in more detail below.

Stateless policies

Stateless policies determine whether or not an operation is permitted based on information associated with that operation alone. For instance, a stateless security policy could specify that a specific function call performed on a specific object is only allowed when the value of the first argument is on a predefined whitelist.

WebJail [ARD+11] is an example of such a stateless policy for securely integrating third-party JavaScript. WebJail classifies security-sensitive operations into nine categories, including DOM access, cookies, external communication, device access, etc., which can be permitted or blocked individually. WebJail policies are based on static whitelists for each of the categories, and can easily be implemented with JSand.

Stateful policies

Stateful policies can accumulate internal security state over multiple calls and use this global state as part of the policy, in addition to the local information made available on each operation request. For instance, a stateful security policy could specify that the use of XHR is allowed as long as no cookies have been read. This type of policy is more expressive than its stateless counterpart, but it is also more complex to specify and more prone to mistakes.

The shadow page in AdJail [LGV10] is another example of internal state that could be accumulated over multiple calls. This page represents a ghost DOM, which is not directly rendered to the user, but allows an advertisement to execute various DOM operations in a confined environment.

Advanced policies

More complex policies can be used to enforce more advanced security properties, such as information flow security. One example of this is a set of policies to implement *noninterference* through *secure multi-execution* (SME) [DP10, DGDNP12]. For any script, SME can classify each input and each output channel as either H (high security, confidential) or L (low security, public). A script is noninterferent if its low-level outputs are not influenced by high-level inputs. Consider for instance the following script on a webserver at `mydomain.com`.

```
1 var cookies = document.cookie;
2 document.getElementById('some-img').src =
3           'http://attacker.com/img.jpg?c=' + escape(cookies);
```

The first line can be classified as H input, since cookie values are security sensitive. The second line can be classified as L output, since this triggers an HTTP request to a different domain. This program is interferent, because the low-level output statement at line 2 is clearly influenced by the high-level input statement at line 1.

Under secure multi-execution, a script is run multiple times, once for each security level. Outputs of a given security level are only generated in the execution belonging to that security level and inputs of a given security level are replaced by `undefined` in all executions of a lower level. Hence, high-level, security-sensitive input can never leak to low-level, public output channels, or even have an influence on them.

To multi-execute a script using JSand, that script must be executed once for the low security level and once for the high security level, each time in a different sandbox, with a different security policy. The low-level policy should disable all high-level inputs and ignore high-level outputs, while the high-level policy should simply ignore low-level outputs. Since each output statement is executed in only one of the executions, the net effect of a noninterferent script under secure multi-execution will be the same as the net effect of executing the same script without multi-execution.

7.4.4 Wrapping the DOM

All interactions between a script and the browser are performed through the DOM. Hence, to control access to all security-sensitive operations, JSand needs to control access to all facets of the DOM. To implement this, each sandboxed script is initially only given a single reference to a wrapper of the `window` object, which is the root of the DOM tree. As described in Section 7.4.2, all property accesses, property assignments and function calls on this wrapper or on any object transitively reached from it are intercepted by a handler. These handlers can thus enforce an arbitrary policy on the entire DOM, and hence effectively control access to all security-sensitive operations.

For any DOM object wrapper, a distinction can be made between two categories of properties. The first category consists of *standard DOM properties*, i.e. properties that are part of the DOM as defined by the ECMAScript standard (or implementation-specific properties provided by the browser). The second category consists of *custom properties* that have been added to a DOM object wrapper by a sandboxed script. For instance, `window.document` belongs to the first category, while `window.googlemaps` could belong to the second. Properties from these two categories need to be handled differently. Assignments to standard DOM properties should be propagated outside the sandbox to the corresponding target property on the real DOM object (if allowed by the security policy), since this is the only way a sandboxed script can interact with the browser. Custom properties should however be confined to the bounds of the sandbox, to prevent sandboxed code from polluting the global namespace and from reading or modifying properties defined outside the sandbox.

To make the distinction between standard DOM properties and custom properties, JSand uses a statically defined *DOM description*, derived from the W3C DOM specification [W3Cd]. This description consists of an array of property descriptors, indexed by a DOM type and a property name. Each descriptor corresponds to a standard DOM property.

7.4.5 Dynamic script loading support

From experience, we have learned that many scripts dynamically load additional scripts during their execution. This is typically accomplished by inserting a new script tag with a `src` attribute in the document, because this method is not under restriction of the same origin policy. However, when a script is included this way, it is executed in the global context. Hence, if we would allow sandboxed scripts to simply add new script tags to the document, they could trivially break out of their sandbox; any script included by a sandboxed script should execute within that same sandbox.

For this reason, JSand uses special handlers to intercept methods that allow script tags to be added to the document, including `node.appendChild`, `node.insertBefore`, `node.replaceChild`, `node.insertAfter` and `document.write`. The first four of these take a (partial) DOM tree as argument and append or insert it at a certain place in the DOM. Our handlers for these methods search the given DOM tree for script tags, extract the value of the `src` attribute and execute the corresponding scripts in the sandbox that included them, using the `innerLoadScript` function described in Section 7.4.2. The `document.write` method is similar but takes an HTML string as argument and appends that string verbatim to the document. The handler for this method parses the given HTML string, extracts script tags out of it and loads them as described above.

We have considered two different techniques for parsing a given HTML string in JavaScript. The first technique consists of creating an iframe and setting its `srcdoc` attribute [Ber12] to the given HTML. To prevent the iframe from fetching and executing scripts included in the HTML, its `sandbox` attribute [HH10] must be set as well. The second technique consists of using a pure JavaScript library to parse the HTML [Joh]. The iframe-based technique has a potential performance benefit, since the parsing is done by native code in the browser instead of in JavaScript. Moreover, using the first technique ensures that the HTML is parsed exactly as the browser will interpret it. However, one of the problems of this approach, is that the parsing is performed asynchronously. That is, we can only access the iframe's fully populated DOM tree from its `onload` callback, which is triggered some time after setting the `srcdoc` attribute. Consequently,

scripts that immediately make use of the HTML written by `document.write` could fail, since the HTML might not yet have been processed. Performing a continuation-passing style transformation on these scripts could solve this problem, but this is a complex transformation which we leave for future work. Our prototype uses the second technique, since it does not suffer from this problem.

7.4.6 Support for legacy scripts

Although the SES library natively supports scripts adhering to recognized ES5 best practices, as described in Section 7.4.1 not all currently deployed JavaScripts do so. Although the secure confinement of legacy scripts is never at stake, not all of them run without errors under the SES library. Therefore, we have developed a support layer to further improve the compatibility with these legacy scripts, based on three abstract syntax tree (AST) transformations.

T1
: Adding a property to the global `window` object normally introduces that property as a global variable, but this does not hold in a SES environment. This transformation introduces a global alias variable for each property of `window`. The variable is updated whenever an assignment is made to its corresponding property.

T2
: Conversely, declaring a global variable normally creates an alias property on the `window` object, but this does not hold in a SES environment. This transformation adds a property on `window` for each global variable. The property is updated whenever an assignment is made to its corresponding global variable.

T3
: Since SES enforces strict mode for all scripts, ES5 non-strict code might be incompatible with SES. The most common incompatibility we have encountered is the lack of `this`-coercion. That is, `this` is no longer bound to the global `window` object in a function call. This transformation replaces `this` by the expression `(this === undefined ? window : this)`.

We have implemented a client-side component for applying these transformations, using the UglifyJS JavaScript parser [Mih]. These transformations do not provide a full translation from ES5 to SES, but they are sufficient to make many legacy scripts work with our prototype.

7.5 Evaluation

In this section we evaluate to what extent JSand satisfies the requirements set forth in Section 7.2.3.

7.5.1 Complete mediation

All sandboxed scripts are executed in an object-capability environment, set up by the SES library. Our implementation of the Membrane pattern ensures that each DOM access and JavaScript API call made by a sandboxed script is assessed by the security policy. Based on the theory of object-capability systems, this provides complete mediation.

Note that JSand provides a one-way isolation and hence makes no attempt to protect a sandboxed script from its environment. That is, code running in the global security context, such as browser plugins and unsandboxed scripts, have the power to modify a sandbox's security policy or to inject a DOM proxy that allows access to any DOM object. However, since malicious global code has already full power over the web page, we consider protecting against such scenarios out of scope for our solution.

7.5.2 Backwards compatibility

We have extensively and successfully tested our prototype on a variety of JavaScripts. In this section we report and discuss in detail three of the most widespread included scripts around: Google Analytics, Google Maps and the jQuery library. Google Analytics is included from more than 68% of all domains from the Alexa Top 10 000, making it the most included script on this list [NIK+12]. Google Maps is the most included web mashup API according to [Pro], being used in 17.41% of registered mashups. jQuery is the most popular JavaScript library in use today, included in more than 57% of the top 10 000 websites to date [Bui]. As future work, we would like to extend our evaluation to more legacy scripts.

Google Analytics

Google Analytics (GA) is a web analytics service that generates statistics about visitors to a website. The GA API allows web administrators to collect custom visitor properties, in addition to the standard properties that are collected by

default (such as referrer and geographical location). The collected statistics can be monitored using a dashboard interface on the GA website.

To enable GA, the website owner must add a small JavaScript code template provided by Google to the header of the page to track. This template sets up an array of options to pass to the GA service and dynamically adds a new script tag to the page to include the main GA script. Any script included like this has unrestricted access to the DOM, making the page vulnerable to malicious script inclusions.

Manual inspection of the GA script is practically impossible, since the code is minified. Moreover, since the main GA script is loaded dynamically from the Google servers, any static, offline security analysis would fail to detect malicious changes introduced to the script after the initial analysis. However, by running GA in a JSand sandbox with a policy that permits only the operations necessary for a benign web analytics script, the impact of a malicious action on behalf of the GA script can be reduced to a minimum. The code snippet below shows how this can be implemented.

```
1  // main page:
2  var sb = new jsand.Sandbox('ganalytics.js',policy);
3  sb.load();
```

This code snippet creates a new sandbox and initializes it with the `ganalytics.js` script, which is shown below and consists of the code template provided by Google.

```
1  // ganalytics.js:
2  var _gaq = _gaq || [];
3  _gaq.push(['_setAccount', 'UA-xxxxxxxx-x']);
4  _gaq.push(['_trackPageview']);

5  (function() {
6    var ga = document.createElement('script');
7    ga.src = 'http://www.google-analytics.com/ga.js';
8    var s = document.getElementsByTagName('script')[0];
9    s.parentNode.insertBefore(ga, s);
10 })();
```

Both the `ganalytics.js` script and the main `ga.js` script (which is loaded from the code above) are executed in the same sandbox and are patched-up automatically, based on the AST transformations described in Section 7.4.6. The following code fragment shows the first two lines of the patched-up `ganalytics.js`.

```
// patched-up ganalytics.js:
var _gaq = _gaq || [];
window._gaq = _gaq;
[...]
```

The global variable `_gaq` is explicitly aliased as a property on `window`. This transformation is necessary because the `ga.js` script frequently refers to the `_gaq` array as `window._gaq`. Such references would fail without the patch shown here.

The `_gaq` array exposes an API to interact with GA after it has been initialized, for instance to add a custom property to collect or to track the click of a button. The website owner can access this array using the `innerEval` method described in Section 7.4.2. To facilitate these interactions and to make abstraction of the fact that GA is running in a sandbox, the website owner could implement an object that automatically forwards its calls to the `_gaq` array inside the sandbox.

Clearly, the effort required to run GA in a JSand sandbox is minimal and introduces no disruptive changes whatsoever. Nevertheless, this power of the GA script is reduced to a safe minimum, dramatically reducing the impact of a malicious script inclusion attack.

Google Maps

The Google Maps (GM) API enables website owners to embed a Google Maps gadget on their website. The standard way to add this gadget to a page is to (1) place a div element somewhere in the body where the map should be displayed, (2) add a script tag to the head of the page, which loads the GM library from the Google servers and (3) add a small piece of JavaScript code to the page, to create a new GM instance in the div element.

As with Google Analytics, the default way of including the GM script lets it have unrestricted access to the DOM and JavaScript APIs, putting the confidentiality and integrity of the entire web page at risk. JSand enables the website owner to confine the GM gadget to a sandbox with the minimal privileges required for legitimate operation.

The steps required to run GM in a JSand sandbox are very similar to the standard steps described above. In step (1), in addition to placing a div element somewhere in the body, the integrator must include the JSand library and the libraries it depends on. In step (2), instead of adding a script tag to directly load the GM library in the global page context, a new sandbox must be created for

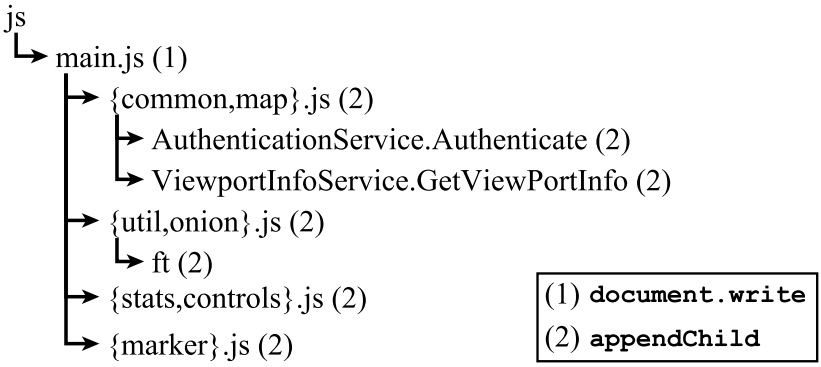


Figure 7.2: Tree of scripts dynamically loaded by Google Maps.

the GM script to run in. In step (3), the website owner can use the `innerEval` method to create a new GM instance in the sandbox. These steps are depicted in the following code fragment.

```
1  var sb = new jsand.Sandbox(
2     'http://maps.googleapis.com/maps/api/js?sensor=false',
3     policy);
4  sb.load();
5  sb.innerEval(
6   "var m = window.google.maps;
7    var options = {
8      center: new m.LatLng(-34.397, 150.644),
9      zoom: 8, mapTypeId: m.MapTypeId.ROADMAP
10   };
11   var map = new m.Map(document.getElementById('map_div'), options);"
12 );
```

When the main GM script is loaded, a complex process of dynamically loading and patching other scripts is performed in the background. Figure 7.2 depicts the sequence of scripts that are dynamically loaded from the main `js` script initially loaded in step (2). In addition to the scripts shown in this figure, more scripts are loaded and patched whenever the user changes the map's viewport (by dragging it or changing the zoom level). All three translations described in Section 7.4.6 are required for the GM gadget to work.

The GM API provides extensive support for customization, to support feature-rich web mashups built around the GM gadget. For instance, website owners can provide custom map overlays, place markers, register callbacks for mouse events, etc. As with GA, a website owner can use the `innerEval` method to interact with the sandboxed GM gadget.

The fact that JSand can successfully execute this gadget in a sandbox without any compatibility issues, illustrates that our solution is able to sandbox complex JavaScript gadgets that depend on dynamic script inclusions and that feature advanced DOM interactions.

jQuery

The jQuery library aims to provide a simple cross-browser API for performing common JavaScript operations, such as creating and selecting DOM elements, handling events, invoking Ajax interactions, etc. While jQuery can be used as an abstraction layer on top of an extensive set of JavaScript APIs, a website owner typically uses only a limited subset of what the library has to offer. By running jQuery in a sandbox with tight restrictions on the permitted JavaScript API and DOM operations, the risk and impact of a malicious script inclusion attack are reduced dramatically.

For our jQuery evaluation scenario, we executed jQuery together with the jQuery-geolocation plugin [NoM] in a sandbox, using a fine-grained security policy that allows us to toggle access to the JavaScript Geolocation API. Disabling the Geolocation API in the policy effectively prevents jQuery from using it in the sandbox. The following code fragment shows how this scenario is implemented.

```
1  var sb = new jsand.Sandbox('jquery-1.7.2.js',policy);
2  sb.load();
3  sb.innerLoadScript('jquery-geolocation-0.1.js');
4  sb.innerEval(
5   "if (jQuery.geolocation.support()) {
6      jQuery.geolocation.find(function(loc) {
7        alert(loc.latitude+\", \"+loc.longitude);
8      });
9    } else { alert('Geolocation not supported'); }");
```

This scenario illustrates that, with minimal effort, a website owner can create a secure JSand sandbox around an extensible JavaScript library, while still being able to interact with it from outside the sandbox.

7.5.3 Performance benchmarks

To evaluate the runtime overhead of our prototype, we have conducted micro- and macro-benchmarks. All benchmarks were run using Google Chrome v20.0.1132.21 on an Intel Core 2 Duo T8300 2.4GHz processor with 4GB RAM.

Micro benchmarks

JSand framework load time. To measure the load time of the JSand framework, a page was created that loads the framework but does not use it. This page was reloaded 1000 times and the elapsed time was recorded. The average load time measured in this way was 71.5±1.8 ms. The same experiment was run with all JavaScript code commented out, so the same network load time would be maintained, but the code would not be executed. The load time in this case was 23.0±0.2 ms. This means that once loaded from the network, the framework takes on average 48.5 ms to deploy on the client side.

Third-party library load time. Similar experiments were performed to measure the overhead of loading and parsing a third-party JavaScript library into a JSand sandbox. We chose jQuery as a representative JavaScript library and loaded it in a JSand sandbox, as well as a regular, unsandboxed JavaScript environment, using XHR and eval(). In both cases, we supplied a real JavaScript library as well as a commented-out version to factor out network overhead.

In a regular JavaScript environment, the code loads in 53.0±0.8 ms and 26.8±0.2 ms for normal and commented-out code respectively. Inside a JSand sandbox, the code loads in 1458.2±16.0 ms and 107.6±1.4 ms respectively, so that the overhead of parsing the library code is about 1350.6 ms.

A large portion of this overhead is due to the script rewriter of the legacy support layer described in Section 7.4.6. Since jQuery is SES-compliant, this rewriting step is not required. Disabling it lowers the average load time from 1458.2 ms to 705.8±1.1 ms, and the average overhead from 1350.6 ms to 598.2 ms. This means that 44.3% of the overhead can be contributed to our efforts for making legacy code SES-compliant.

Membrane transition cost. To verify the runtime overhead of a function call crossing the membrane, a function was executed both inside and outside a JSand sandbox 1 million times and the elapsed time is recorded. We chose the `window.clearTimeout` function as a representative function, because intuitively it should return quickly when no timer is registered. When called from inside the sandbox, the `window.clearTimeout` call must cross the membrane separating the sandbox from the real JavaScript environment. Outside the sandbox, the average execution time is 0.9±0.0 μs, while inside the sandbox it is 8.0±0.1 μs.

Macro benchmarks

The most important metric that counts when executing JavaScript in a browser, is the user experience. Ideally, the user should not notice that JSand is being used at all. To measure how much overhead the user experiences, we created a typical web application using Google Maps and measured two things: the total load time of the web application, and the delay a user experiences when interacting with it.

The load time of the web application is measured from the time the page is loaded until the Google Maps API emits a 'tilesloaded' event, signaling that the application is ready to be used. Running outside of the JSand sandbox, this load time is 308.0±13.7 ms, and 1432.8±24.2 ms inside of it. Keeping in mind that a large portion of this overhead is due to script-rewriting for legacy code, the total overhead without the legacy support layer can be estimated to be about 626.5 ms.

To measure the delay experienced when interacting with the application, we wait until the application is loaded, and then pan 400 px to the right, 100 times. The average time elapsed between 2 pans is considered as a reasonable approximation of the user-experienced delay. This delay is 320.2±0.8 ms outside and 420.0±2.7 ms inside the sandbox.

The overall performance of a JSand sandbox is acceptable. The overhead when loading a reasonably-sized SES-compliant JavaScript library inside the sandbox, is about 203%. For legacy scripts, JSand requires a code transformation step that results in a total overhead of about 365%, but it is expected that this step can be removed or at least sped up significantly for future JavaScript code in future browsers. Furthermore, the tendency by users to keep certain websites open using persistent tabs, makes the load time overhead less important. Additionally, despite the nine-fold execution time of a function-call traversing the sandbox membrane, the delay experienced by a user when using a realistic web application inside a JSand sandbox, is a quite acceptable 31.2%, corresponding to an absolute delay on the order of 100 ms.

7.6 Related work

Server-side processing of scripts. A common technique for preventing undesired script behavior is to restrict the untrusted code (i.e. the third-party component) to a safe subset of JavaScript [MT09]. Compliance to the subset is verified at the server side. The allowed operations within the subset prevent the untrusted code from obtaining elevated privileges, unless explicitly allowed

by the integrator. ADSafe [Cro], ADsafety [PEGK11] and FBJS [The] are examples of techniques where third-party JavaScript must conform to a certain JavaScript subset. Techniques such as Caja [MSL+08], Jacaranda [Jac] and Live Labs' Websandbox [Micj] on the other hand, statically analyze and rewrite the third-party JavaScript on the server side into a safe version.

Instead of forcing the use of a JavaScript subset, the JavaScript code can also be instrumented with extra checks that mediate access to certain functionality. BrowserShield [RDW+06] and Browser-Enforced Embedded Policies (BEEP) [JSH07] are examples of such instrumentation on the server-side.

While safe subsets, code rewriting and server-side code instrumentation can restrict third-party code at the source, their adoption by mashup integrators is problematic. These techniques require either access to code running on the server-side, or require the website owner to implicitly trust the JavaScript provider to deliver safe JavaScript code. In real-world scenarios, it is infeasible to impose any such restrictions on third-party code providers. In contrast, JSand requires no server-side processing of the third-party code and imposes no fundamental restrictions on included code.

Extending the browser with a reference monitor. A second class of techniques extends the browser to enforce code restrictions. Systems like ConScript [ML10], WebJail [ARD+11] and Contego [LD11] require modifications to the JavaScript engine to enforce policies on third-party code, while AdSentry requires the installation of a Firefox extension to restrict the functionality available to advertisements.

Browser modifications to restrict third-party JavaScript can be implemented very efficiently and can guarantee that enforcement cannot be circumvented. The major disadvantage of this approach however, is that the browser must be modified. Unless all the users of a web application are using a browser which implements the desired modification, there is little or no incentive for the website owner to make use of it. Because of the large variety of active browser vendors and versions on the internet, it is unrealistic to assume that a certain modification will ever be implemented in all browsers. For this reason, JSand does not depend on any special browser-side features except for what is available in the web standards.

Leveraging existing browser security features. Finally, some approaches leverage recent browser security extensions to contain scripts. The new `sandbox` attribute of the iframe element in HTML5 [Ber12] can restrict third-party

JavaScript in a very coarse-grained way: it only supports to completely enable or disable JavaScript.

The Content Security Policy (CSP) [SSM10] allows the insertion of a security policy through HTTP response headers and meta tags, which must be enforced in the browser. This policy can restrict the locations a web application loads its content from, thus preventing some forms of content-injection. Unfortunately, CSP does not provide any fine-grained control over which JavaScript functionality is available to restricted code.

AdJail [LGV10] is geared towards securely isolating ads from a hosting page for confidentiality and integrity purposes, while maintaining usability. The ad is loaded on a shadow page that contains only those elements of the hosting page that the web developer wishes the ad to have access to, and it relies on the SOP to isolate the shadow page. Changes to the shadow page are replicated to the hosting page if those changes conform to a specified policy. Likewise, user actions on the hosting page are mimicked to the shadow page if allowed by the policy. AdJail is a good approach to restrict access to the DOM, but cannot enforce a policy on the other JavaScript APIs like JSand does.

Self-protecting JavaScript (SPJS) [PSC09, MPS10] is a client-side wrapping technique that applies advice around JavaScript functions, without requiring any browser modifications (unlike [ML10] or [ARD+11]). It builds on standard aspect-oriented libraries for JavaScript. The wrapping code and advice are provided by the server and are executed first, ensuring a clean environment to start from. SPJS does not guarantee that all access-paths to certain JavaScript functionality can be restricted, because the aspect library it relies on was not designed with security in mind. JSand uses the Membrane pattern instead. This pattern was designed to provide complete mediation.

Secure ECMAScript (SES) [Mil] is a subset of ES5 strict which provides an object-capability language. Unlike Caja, from which it originated, SES runs completely on the client-side without any browser modifications. To the best of our knowledge, JSand is the first fully functional JavaScript integration technique built on SES, capable of handling legacy scripts such as Google Maps and Google Analytics.

7.7 Conclusion

This chapter introduced JSand, a server-driven but client-side JavaScript sandboxing framework that does not rely on any browser modifications. We have implemented a prototype of this framework and evaluated it on the most

widespread JavaScripts around. Although there has been a lot of activity in this research area, we are the first to deliver a solution that provides complete mediation, backwards compatibility and an acceptable performance overhead.

Acknowledgments

This research is partially funded by the Research Fund KU Leuven, the EU-funded FP7 projects NESSoS and WebSand and by the IWT-SBO project SPION. Pieter Agten holds a Ph.D. fellowship of the Research Foundation - Flanders (FWO). With the financial support from the Prevention of and Fight against Crime Programme of the European Union European Commission - Directorate-General Home Affairs. This publication reflects the views only of the author, and the European Commission cannot be held responsible for any use which may be made of the information contained therein.

Chapter 8

Conclusion

"The world is changed by your example, not by your opinion"
— Paulo Coelho, writer

The web is a marvelous piece of technology connecting billions of people across the planet through a wide range of web applications. JavaScript is one of the main components that makes these modern web applications possible. Unfortunately, the usage of JavaScript has introduced the web to some new vulnerabilities, specific to web applications and web browsers. Badly sanitized input can lead to malicious JavaScript being leaked into a JavaScript execution environment, where it is then executed unintendedly. Untrusted JavaScript, sometimes originating from compromised servers, included in a web application, is intentionally executed in a trusted environment. Both unintended and intended execution of untrusted JavaScript can give attackers control over a web application's resources and functionality.

This work focuses on isolating and restricting untrusted JavaScript in order to take away a web application's sensitive resources and functionality from attackers.

The goal of this work was two-fold:

- First, study ways through which JavaScript code can leak into the JavaScript execution environment of a browser, executing unintendedly.
- Second, isolate and restrict untrusted JavaScript code into a JavaScript sandbox, no matter whether or not it was intended to be executed.

In this concluding chapter, Section 8.1 reviews the contributions in this work, Sections 8.2 and 8.3 lists some lessons learned about JavaScript sandboxing and large-scale experimentation on the Internet, Section 8.4 looks at some potential future work and Section 8.5 concludes with some concluding thoughts.

8.1 Contributions

This work contributes to the first goal in Chapters 3 and 4.

In Chapter 3, we studied JavaScript leaking into the web browser through browser plugins such as the Flash plugin. We built FlashOver to automatically detect XSS vulnerabilities in Flash applications using a combination of static analysis and automated interaction. FlashOver was then evaluated on almost 15,000 Flash application from the Web's most popular 1,000 domains according to Alexa, and found vulnerable Flash applications on some of the Web's most popular websites. This work shows that JavaScript can indeed leak into the browser through plugins when input is badly sanitized.

In Chapter 4, we studied JavaScript leaking into the web browser through browser extensions such as the Greasemonkey extension. We investigated the Greasemonkey extension for vulnerabilities and statically analyzed more than 86,000 user scripts from userscripts.org, Greasemonkey's official script market, for malware and vulnerabilities. We discovered that JavaScript leaking into the Greasemonkey extension can get access to very powerful functionality that bypasses the same-origin policy, the foundation of security on the Web. This work shows that JavaScript can indeed also leak into the browser through extensions, again when input is badly sanitized.

The second goal of this work requires JavaScript sandboxing. Chapter 5 studies the research related to JavaScript sandboxing and divides it into three categories:

- JavaScript sandboxing using JavaScript subsets and rewriting systems,
- JavaScript sandboxing through browser modifications and
- JavaScript sandboxing without browser modifications.

This work contributes to the second goal in Chapters 6 and 7, by building, studying and evaluating two JavaScript sandboxing systems.

In Chapter 6, we studied JavaScript sandboxing through browser modifications. We built WebJail, a browser modification in Firefox, which allows a web developer to specify a fine-grained policy over an integrated mashup component

in an iframe. WebJail allows the restriction of third-party JavaScript code by binding all sensitive DOM functionality with advice functions that mediate access to that functionality. This work shows that it is possible to sandbox JavaScript using a browser modification, but the resulting modified browser is difficult to maintain and distribute to end-users.

In Chapter 7, we studied JavaScript sandboxing without browser modifications. We built JSand, a JavaScript library that leverages Secure ECMAScript (SES), a standardized JavaScript subset which forms an object-capability environment, and the new JavaScript Proxy API, to wrap the DOM and all its sensitive functionality according to the membrane pattern. JSand provides backward compatibility with legacy code that does not conform to the Secure ECMAScript subset by rewriting it with a built-in JavaScript rewriting layer. JSand was evaluated on some mainstream web applications and works in any modern browser. This work shows that it is possible to sandbox JavaScript without browser modifications, removing both the maintenance and distribution limitations of JavaScript sandboxing systems that do require browser modifications. The performance is acceptable but not yet optimal. We feel that the future will optimize these new technologies and introduce even better ones to help with JavaScript sandboxing.

8.2 Lessons learned: JavaScript sandboxing

The unique architecture of the Web, with regard to the usage of JavaScript in web applications, gave rise to equally unique client-side vulnerabilities and attacks.

Bad input validation and misplaced trust are known to cause security issues, but they also present new challenges on the web. When untrusted JavaScript code finds its way into a JavaScript execution context, whether it be intended or unintended, it should be restricted to limit the potential damage it can cause in a web application, by executing it in a JavaScript sandbox instead.

JavaScript subsets and rewriting systems are a promising way to solve the problem by trying to force untrusted JavaScript to use a JavaScript subset that can more easily be verified. Non-conforming code is rewritten to add extra runtime instrumentation so that a policy can be enforced at runtime. Unfortunately, these systems have a deployment problem. A middlebox in which such a system is implemented will change the architecture of the Web if it is hosted on a remote web server. Placing the middlebox at the client-side solves that problem, but forces users to run additional software next to their browser.

Another option would be a browser modification that implements JavaScript sandboxing logic. Such a modification has direct and unrestricted access to the JavaScript environment, allowing it to mediate all access by untrusted JavaScript. Modifying a browser is not desirable because it requires users to install a special browser, and the modification must be maintained with new browser versions.

Both for maintenance and distribution reasons, a solution that does not require browser modifications is better in the long run. Unfortunately, JavaScript sandboxes built with existing technologies are limited to the available functionality and that functionality's performance.

For research, creating a browser modification for JavaScript sandboxing is a necessary evil in the short run. The work however, should not stop there. To be useful in the long run, any new browser functionality that leads to an efficient and successful JavaScript sandboxing system, should be standardized. The standardization process will ensure that this new JavaScript sandboxing system becomes available in all browsers, and thus to all current and future users of the Web.

In this work, we looked at the isolation and restriction of JavaScript code, which in itself provides many challenges. The field of *information flow control* can be seen as an extension on access control. Instead of only tracking the execution of code, it also tracks the propagation of information from different sources through a system and limits what information is exposed. For instance, information from public and private sources is tracked inside a system. Combining information from both sources is allowed as long as the result is not exposed to a public channel. Research in this field faces equally challenging questions that are beyond the scope of this work. For more information, we refer to the work of other researchers [SM03, DP10, DGDNP12, Raf14, SYM+14].

In addition to isolating and restricting JavaScript in generic web browsers, this work can be of value for specialized web browsers as well. Mobile applications, commonly found on smart phones, have a need to interact with the Web. To help with this task, mobile application developers can embed a software component into their application to display web pages. One such component is called the WebView component on Android and it exists on other platforms by similar names. Through WebView, a mobile application can offer extra functionality to the JavaScript environment associated with the loaded web page. Without proper care, mobile application developers making use of this feature, can introduce security vulnerabilities allowing attackers to take control of the JavaScript execution environment. Chin et al. [CW13] found that 70.3% of Android applications make use of WebView and that 11% of those are vulnerable to outside attackers. Following similar reasoning as with vulnerable browser

plugins and browser extensions, it would be beneficial to mobile application development to be able to restrict what functionality is available to untrusted JavaScript code.

8.3 Lessons learned: Large-scale experimentation on the Internet

The Internet consists of billions of interconnected computers. The Web consists of billions of web sites [Netc] and trillions of web pages [Good], and it keeps growing every day.

Without lifting the limitations of current methods to study the Internet and the Web, we will quickly lose oversight. Studying emerging properties and threats on the web requires an ability to massively gather and process data, which in turn requires a distributed computing setup that can scale to accommodate the growth of both Internet and Web.

We [Vanar] show that it is possible to build a large-scale experimentation setup with limited access and resources, and that it can be used to gather intriguing research results. A large-scale experimentation setup is a software system that, in our case, faces the following challenges and requirements:

Deployability.
The software must be easily deployable on the hardware and require no or little special infrastructure or privileges to run. This applies to anything that requires superuser permissions, or any infrastructure that needs to be installed by a superuser.

Security.
The software must take security into consideration. Any network facing services should require authentication and all network communication should be encrypted. Even with authentication and encryption, network facing services can have exploitable bugs which affect security on a public network.

Scalability.
A large-scale experiment needs the ability to execute many tasks. It is desirable that these tasks can be executed in parallel on as many processors as possible. Therefor, the software must be able to scale well when a large amount of resources is available.

Storage.
Large-scale experiments produce a lot of data that must be stored somewhere persistently at a reasonable speed, so that storing results does not become a bottleneck during the experiment. Typically, this means that the storage facility must be able to handle at least several Terabytes of data, and have sufficient bandwidth to accommodate hundreds of concurrent tasks writing and reading data.

Resilience.
The software system should be resilient against failures of both nodes and tasks. When a node becomes unavailable or disappears, any task running on it should be repeated. Likewise, when a task temporarily fails, it should be repeated if desired.

Maintainability.
The software should be maintainable so that improvements and bugfixes can be applied. For custom software this can often be a problem when the original author becomes unavailable. Publicly available software should have an active support and development community.

Usability.
The software should be flexible enough to conduct different kinds of experiments, and it should be reasonably easy to use, or have a non-steep learning curve together with enough documentation and examples.

During the course of this PhD project, three large-scale experimentation frameworks were developed that meet these requirements in various degrees. Each of them was created to conduct and support various research projects, such as EU-FP7 WebSand [weba], EU-FP7 STREWS [str], EU-FP7 NESSoS [nes], EU-ISEC B-CCENTRE [bccb], BELSPO-BRAIN BCC [bcca], iMinds-ICON TRUBLISS [tru], IWT-SBO SEC SODA [sod] and IWT-SBO SPION [spia], requiring the analysis of a large amount of specific Internet or Web resources.

Mjolnir is a robust large-scale experimentation framework that is generic and easy to deploy. As the earliest of the three, Mjolnir has been used in many large-scale experiments of published work [NIK+12, AND+12, AND+14, CNHD13, vGCN+14, DRND+12] and unpublished work to analyze web pages, Flash applications, JavaScript code, cookies and more.

Boreas is more resilient and scalable than Mjolnir, but less deployable because it uses a VPN for internal communication. Boreas was mainly used in unpublished research to determine the usage of VBScript on the Web. It was also used to gather empirical evidence about the sizes of “Set-Cookie” HTTP headers, in support of research of colleagues [DR14].

The ZergHive is even more resilient and scalable than both Mjolnir and Boreas. It is also easier to maintain and use because it reuses existing components such as the Python Celery distributed task queue and an AMQP middleware for communication. Because it requires more components, it is less deployable. The ZergHive has been used to crawl millions of web pages on a quarterly basis, daily crawls to gather historical data for a project, and analyzing cross-domain requests for as of yet unpublished research.

Building such homebrew frameworks can be a time-consuming process that involves many iterations due to unforeseen problems and limitations. Overcoming these problems and limitations is a valuable experience that helps to gain insight and better understand engineering challenges faced when scaling up to "large scale."

If learning about these engineering challenges is not the main goal of setting up a large-scale experimentation setup, it is best to consider alternative, existing solutions. Solutions such as Hadoop are used in the real world by major players on the Internet, and promise dazzling performance. The downside is that such solutions likely require a specialized environment in which they can operate, and that more time needs to be invested in their setup.

We [Vanar] also list some lessons learned from conducting large-scale experiments on the Internet and the Web, including a motivating "story from the trenches," showing how a simple oversight during research for FlashOver, can lead to an undesired and problematic, although hilarious, outcome.

Other lessons and advice learned from large-scale experimentation include:

- Limit task and worker capabilities to reduce potential damage when an experiment goes wrong.
- Secure nodes to avoid that they are compromised by attackers while conducting an experiment with untrusted code or resources.
- Reduce bottlenecks and expect bottlenecks to move to unexpected places.
- Provide a clear point of contact for outsiders, in case an experiment goes wrong that has influence on the outside world.
- Write robust scripts to analyze untrusted data that can handle the unexpected input found surprisingly often on the Internet.
- Expect components out of your control to break.
- Perhaps the most important: earn the trust and respect from the support staff.

8.4 The future

If there is one thing we can learn from the past, it is that we have been inaccurate when predicting the future. Visions of the future from the past have promised hover cars, jet packs and a world without human suffering, where we have colonized space and all mundane tasks are handled by artificially intelligent robots. While some of these visions are close to today's reality, they took longer to unfold than anyone had thought, and not necessarily in the direction we were all dreaming they would.

Still, after spending four years on this PhD project, I believe I have gained a unique view on this field and there are some things that I think are within our grasp and should be relatively easy to implement.

In addition, I would like to consider a wider view on Web security farther into the future and point out some bigger issues that need solving.

Short term

Firefox evalInSandbox() with a wrapped global object. Firefox offers an evalInSandbox() function to browser extensions like Greasemonkey, through which these extensions can execute JavaScript in a specified origin and with a specified global object. This function is implemented in native code in SpiderMonkey, Firefox's JavaScript engine, and thus performs really well. Although this function is only available in Firefox, it offers the basis for a simple sandboxing mechanism, since it provides isolation into an origin and the ability to specify a custom global object.

If it were possible to wrap the DOM of a webpage using the membrane pattern, as was done in JSand, and pass this wrapped DOM as the global object to evalInSandbox(), this would result in a lightweight JavaScript sandbox that can be implemented as a Firefox extension. Unfortunately, while attempting such implementation myself, I have discovered that evalInSandbox() triggers some kind of runtime exception when a Proxy object is passed to it. The first step in the implementation of this lightweight sandbox would thus be to make evalInSandbox() accept a Proxy object.

SandboxWorker: a WebWorker extension to access a wrapped DOM. TreeHouse uses WebWorkers to isolate JavaScript code in their JavaScript sandboxing system. WebWorkers do not have access to a webpage's DOM, but instead have a minimalistic DOM with only a few methods available, such as postMessage(). TreeHouse communicates with its WebWorker through

postMessage(), and marshals any access to the DOM of the webpage through this communication channel, which is an expensive process.

Instead of limiting a WebWorker to a minimalistic DOM, it can be beneficial to create a version of a WebWorker, let's say a SandboxWorker, where the global object can be configured. This would allow, just like in the previous idea, a wrapped global object to be used as the DOM, instead of a fixed minimalistic DOM. The current standardized WebWorker would then be a special case of such a SandboxWorker, allowing for backward compatibility.

Surfing the Web in the past, through a proxy. Something all large-scale experiments on the Web have in common, is that they download and process a lot of data. Conducting many large-scale experiments consumes a lot of bandwidth to retrieve the data, a lot of CPU power to process it and a lot of disk space to store the results. A more optimal approach would be to gather the data needed for any large-scale experiment into a cache once, and reuse it for all experiments in that time-frame.

Such a cache could be implemented using a web-proxy. The data gathering process only needs to download all required resources through the proxy, which will cache it. Subsequent large-scale experiments can then be conducted through that proxy as well, reusing the data and saving bandwidth. When collected on a regular basis, with a cache for e.g. every month, such a cache collection can become the basis for a "time machine." Experiments can be repeated at a later time by downloading and processing old data through the time machine. The results should then also be identical.

The Internet Wayback Machine does something similar, but does not track all resources. It can also not be operated through a proxy and translates URLs in webpages to be prefixed with the time machine's hostname, thus mangling the original data.

Sandboxing other client-side scripting languages. This work looks at how JavaScript can leak into a JavaScript context and how it can then be isolated and restricted in a JavaScript sandbox. The original HTML 4 specification [W3Cg] allows for more scripting languages than just JavaScript in the browser. Namely, it also lists "text/tcl" and "text/vbscript" as possible alternatives.

As far as we are aware, no research has been done to chart the usage of these other kinds of scripting languages on the Web. Browsers supporting these other scripting languages can also benefit from a sandbox to both isolate and restrict these scripting languages.

Long term

Zero-effort security measures. In the past decades, the Internet and the Web have evolved so much that it is difficult to predict what they will look like in the future. Most likely, both will undergo changes that we can not envision at the moment. However, it is reasonable to assume that neither the Internet nor the Web are likely to disappear.

The Web was built as a documentation system and a collaborative workspace. It has evolved into a global communication system to share information and ultimately bring people closer together. The Web is about people. People do not care about code, they care about other people and sharing information with them, in whatever way.

The typical Internet user does not understand all the underlying technology and we should not expect them to. This also implies that nobody should expect the typical Internet or Web user to make the right choices when it comes to protecting their computer, software or data.

Unfortunately, today's reality is very different. Users are often confronted with security settings and dialogs, forcing them to make a choice between being secure and getting work done. In such a scenario, the user will go for the option that requires the least effort. In most cases today, that is the insecure option.

A fine example is prompting an end-user when encountering an expired SSL certificate: Should the user stop visiting their favorite website due to a perceived network-glitch? Or should the user blissfully ignore the annoying security warning and visit the website anyway, with potentially bad consequences?

Asking technical questions of an end-user is not a good approach to security. Where user settings or user interaction is required, the software developers of the involved software must take human nature into account and assume the end-user to make whatever setting or choice that allows them to continue with their lives. Whatever this setting or choice is, *that* must offer the most security. The zero-effort choice should be the secure choice.

Of course, in the ideal case, end-users should not be prompted with security dialogs at all. Software should be smart enough to know what the secure option is and have that as the default. Any security measure developed in the future should be fully automatic, transparent and work out-of-the-box with secure default settings.

Browsers as the new end-user operating system. In the past, "old-school" applications needed to be compiled into machine code for several CPU

architectures and operating system platforms, and then distributed to end-users on physical installation media.

These days, applications can be compiled to JavaScript to run inside any browser, thus eliminating the need to support multiple platforms. Because web applications can be delivered over the Web directly to the browser, there is no more need to distribute software to end-users on physical installation media. JavaScript can truly be called the assembly language of the Web.

Mobile devices such as smartphones are connected to the Internet and the Web around the clock. Mobile applications, just like old-school applications, must be developed and maintained for several platforms. Because many mobile devices are constantly connected to the Web, it is reasonable to assume that mobile applications will also be replaced by a variation of web applications in the future.

Examples of applications that now exist as web applications are games (e.g. the Unreal Engine has been ported to JavaScript running in a browser), office productivity tools (e.g. Google and Microsoft offer an online word processor, spreadsheet and presentation software), industrial design tools (e.g. AutoDesk offers a version of its 3D design software that runs in a browser), etc. The list of applications that runs in the browser keeps on growing.

In recent years, we have seen how new and powerful browser functionality was added to support the creation of ever-more complicated web applications. Web applications have almost the same capabilities as an old-school desktop application. We can expect browsers to be augmented with extra functionality until the capabilities of web applications have completely caught up.

In the future, we may see a point where browsers, in whatever form, are blended into the underlying operating system. Browsers then become the new operating system on which web applications run.

Proof-carrying code and dynamic analysis of JavaScript. Making the distinction between good software and malware today, often boils down to verifying a signature of downloaded software. Software that carries a valid signature from a trusted software provider, is considered safe. Unfortunately, signed software is no guarantee that the software is harmless. Malware authors are increasingly making use of stolen signing certificates to make their malware seem like trusted software.

Likewise, anti-virus software scans software for signatures of known viruses and triggers an alert when such a signature is found. Unfortunately, anti-virus software is not a bullet-proof guarantee that software is safe either.

A problem in both cases, signed software and anti-virus scanners, is that the end-user needs to trust the opinion of a third party. In case of signed software, the end-user needs to trust that the signing certificate is used with care. In case of anti-virus scanners, the end-user needs to trust that virus definition have been updated and that they can actually detect the latest malware.

A better approach is to use proof-carrying code. Proof-carrying proof carries with it a formal proof of certain properties of the code. The end-user using this code can independently verify these properties by examining the proof in an automated way. The properties proved by this proof can indicate the intentions of the code it is included with. If these intentions do not match the end-user's expectations, then the code can be considered malware and rejected. Otherwise, it can be considered safe.

Because of the dynamic nature of JavaScript, proof-carrying code is not a complete solution for web applications. JavaScript code in a web application can not be guaranteed to be safe before it is executed. Therefor, to guarantee that JavaScript code is not malware, it must be executed and monitored at runtime. The work in this text has contributed towards this goal, but the available tools in web browsers are still lacking.

More than that, the dynamic analysis of JavaScript code should also cover the flow of information during execution of that code. To prevent code with access to sensitive data from leaking that data, that data must be tracked with information-flow techniques. Code that appears to be harmless at first glance, might be leaking this sensitive information to third parties in non-obvious ways.

In the future, browser vendors should build in functionality that aids the dynamic analysis of JavaScript running in a browser, as well as the ability to restrict what functionality and data is available to the executing JavaScript.

8.5 Concluding thoughts

With this text, I hope to have given some small insight into the world of web applications and web browsers, including some of the dangers linked to their use and possible solutions.

Four years ago, when I started this PhD project, I did not know much about web technology. I believed that browsers were opaque software components, JavaScript was no more than an annoying language for popup-advertising and I had never heard of the same-origin policy. Quite honestly, soon after becoming aware of all these technologies and understanding the threats linked to them,

there was a fraction of a millisecond where I considered never using the Web again.

Over time and through better understanding of the issues, most of my concerns have disappeared. With the growth of the Web, Web technologies are continuously created and improved. And although this process also continuously introduces new security problems, they are being addressed.

As of this writing, the Internet has celebrated its 45th birthday and the Web has celebrated its 25th birthday. Looking back to the birth of the Web, it is incredible what has been achieved in such a brief period of human history. The path that brought us here continues on and although we do not know what is over the horizon, I believe that path will go on for quite a while.

Bibliography

[adb] AdBlock Plus. https://adblockplus.org/.

[Adoa] Adobe. About naming variables. http://help.adobe.com/en_US/AS2LCR/Flash_10.0/help.html?content=00000047.html.

[Adob] Adobe. ActionScript 3.0 - Controlling access to scripts in a host web page. http://livedocs.adobe.com/flex/3/html/help.html?content=05B_Security_14.html.

[Adoc] Adobe. ActionScript Technology Center | Adobe Developer Connection. http://www.adobe.com/devnet/actionscript.html.

[Adod] Adobe. Creating more secure SWF web applications. https://www.adobe.com/devnet/flashplayer/articles/secure_swf_apps.html.

[Adoe] Adobe. Flash CS4 Professional ActionScript 2.0 - getURL function. http://help.adobe.com/en_US/AS2LCR/Flash_10.0/help.html?content=00000564.html.

[adof] Flash Player | Adobe Flash Player 11 | Overview. http://www.adobe.com/products/flashplayer.html.

[Adog] Adobe. Flash to focus on PC browsing and mobile apps; Adobe to more aggressively contribute to HTML5. http://blogs.adobe.com/conversations/2011/11/flash-focus.html.

[ale] Alexa - Top Internet Sites. http://www.alexa.com/topsites.

[AND+12] Steven Van Acker, Nick Nikiforakis, Lieven Desmet, Wouter Joosen, and Frank Piessens. FlashOver: automated discovery of cross-site scripting vulnerabilities in rich internet applications. In Heung Youl

Youm and Yoojae Won, editors, *7th ACM Symposium on Information, Compuer and Communications Security, ASIACCS '12, Seoul, Korea, May 2-4, 2012*, pages 12–13. ACM, 2012.

[AND+14] Steven Van Acker, Nick Nikiforakis, Lieven Desmet, Frank Piessens, and Wouter Joosen. Monkey-in-the-browser: malware and vulnerabilities in augmented browsing script markets. In Shiho Moriai, Trent Jaeger, and Kouichi Sakurai, editors, *9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '14, Kyoto, Japan - June 03 - 06, 2014*, pages 525–530. ACM, 2014.

[ARD+11] Steven Van Acker, Philippe De Ryck, Lieven Desmet, Frank Piessens, and Wouter Joosen. WebJail: least-privilege integration of third-party components in web mashups. In Robert H'obbes' Zakon, John P. McDermott, and Michael E. Locasto, editors, *Twenty-Seventh Annual Computer Security Applications Conference, ACSAC 2011, Orlando, FL, USA, 5-9 December 2011*, pages 307–316. ACM, 2011.

[AVAB+12] Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H Phung, Lieven Desmet, and Frank Piessens. JSand: Complete client-side sandboxing of third-party JavaScript without browser modifications. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 1–10. ACM, 2012.

[Bai10] Mike Bailey. Neat, new, and ridiculous flash hacks. In *BlackHat DC*, 2010.

[Bar] Adam Barth. Chromium Blog: Security in Depth: New Security Features. http://blog.chromium.org/2010/01/security-in-depth-new-security-features.html.

[Bar12] Barracuda Labs. When good sites go bad. http://www.barracudalabs.com/goodsitesbad/, 2012.

[bcca] BELSPO-BRAIN BCC. http://www.belspo.be/belspo/fedra/proj.asp?l=nl&COD=BR/132/A4/BCC.

[bccb] EU-ISEC B-CCENTRE. https://www.b-ccentre.be/.

[BEK+10] Marco Balduzzi, Manuel Egele, Engin Kirda, Davide Balzarotti, and Christopher Kruegel. A solution for the automated detection of clickjacking attacks. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 135–144, 2010.

[Ber12] Robin Berjon. W3C HTML5 Working Draft. http://www.w3.org/TR/html5/, September 2012.

[BFSB10] Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman. Protecting browsers from extension vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2010, San Diego, California, USA, 28th February - 3rd March 2010*. The Internet Society, 2010.

[BHS+10] Prithvi Bisht, Timothy Hinrichs, Nazari Skrupsky, Radoslaw Bobrowicz, and V. N. Venkatakrishnan. NoTamper: automatic blackbox detection of parameter tampering opportunities in web applications. In *Proceedings of the 17th ACM conference on Computer and Communications Security*, CCS '10, pages 607–618, New York, NY, USA, 2010. ACM.

[BHvOS12] Joseph Bonneau, Cormac Herley, Paul C. van Oorschot, and Frank Stajano. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, pages 553–567. IEEE Computer Society, 2012.

[BJM09] Adam Barth, Collin Jackson, and John C. Mitchell. Securing frame communication in browsers. *Communications of the ACM*, 52(6):83–91, 2009.

[BKMW10] Sruthi Bandhakavi, Samuel T. King, P. Madhusudan, and Marianne Winslett. Vex: Vetting browser extensions for security vulnerabilities. In *USENIX Security Symposium*, pages 339–354. USENIX Association, 2010.

[Bla10] Dionysus Blazakis. Interpreter exploitation. In Charlie Miller and Hovav Shacham, editors, *4th USENIX Workshop on Offensive Technologies, WOOT '10, Washington, D.C., USA, August 9, 2010*. USENIX Association, 2010.

[Bli] Blink. Blink. http://www.chromium.org/blink.

[Bui] BuiltWith. jQuery Usage Statistics. http://trends.builtwith.com/javascript/jQuery.

[BV08] Prithvi Bisht and V. N. Venkatakrishnan. XSS-GUARD: precise dynamic prevention of cross-site scripting attacks. In Diego Zamboni, editor, *Detection of Intrusions and Malware, and Vulnerability Assessment, 5th International Conference, DIMVA*

2008, Paris, France, July 10-11, 2008. Proceedings, volume 5137 of *Lecture Notes in Computer Science*, pages 23–43. Springer, 2008.

[CCVK11] Davide Canali, Marco Cova, Giovanni Vigna, and Christopher Kruegel. Prophiler: A fast filter for the large-scale detection of malicious web pages. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pages 197–206, New York, NY, USA, 2011. ACM.

[CDP13] Damien Cassou, Stéphane Ducasse, and Nicolas Petton. SafeJS: Hermetic Sandboxing for JavaScript. 2013.

[cera] CERN. http://home.web.cern.ch/.

[CERb] CERN. The World Wide Web project. http://line-mode.cern.ch/www/hypertext/WWW/TheProject.html.

[Cha] Charles Severance. JavaScript: Designing a Language in 10 Days. http://www.computer.org/csdl/mags/co/2012/02/mco2012020007.html.

[chr] Match Patterns - Google Chrome. https://developer.chrome.com/extensions/match_patterns.html.

[CKV10] Marco Cova, Christopher Krügel, and Giovanni Vigna. Detection and analysis of drive-by-download attacks and malicious JavaScript code. In Michael Rappa, Paul Jones, Juliana Freire, and Soumen Chakrabarti, editors, *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*, pages 281–290. ACM, 2010.

[CLR+12] Yinzhi Cao, Zhichun Li, Vaibhav Rastogi, Yan Chen, and Xitao Wen. Virtual browser: a virtualized browser to sandbox third-party JavaScripts with enhanced security. In Heung Youl Youm and Yoojae Won, editors, *7th ACM Symposium on Information, Compuer and Communications Security, ASIACCS '12, Seoul, Korea, May 2-4, 2012*, pages 8–9. ACM, 2012.

[CLZS11] Charlie Curtsinger, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. ZOZZLE: Fast and Precise In-browser JavaScript Malware Detection. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 3–3, Berkeley, CA, USA, 2011. USENIX Association.

[CM10] Tom Van Cutsem and Mark S. Miller. Proxies: design principles for robust object-oriented intercession APIs. pages 59–72, 2010.

[CNHD13] Ping Chen, Nick Nikiforakis, Christophe Huygens, and Lieven Desmet. A dangerous mix: Large-scale analysis of mixed-content websites. In *Proceedings of the Information Security Conference (ISC)*, pages 9–9, November 2013.

[Com] Computerworld. Teen uses worm to boost ratings on MySpace.com. http://www.computerworld.com/article/2558730/malware-vulnerabilities/teen-uses-worm-to-boost-ratings-on-myspace-com.html.

[Con] Web Application Security Consortium. Web Hacking Incident Database. http://projects.webappsec.org/Web-Hacking-Incident-Database.

[Cro] Douglas Crockford. ADsafe – making JavaScript safe for advertising. http://adsafe.org/.

[Cur] Cure53. Flashbang. https://github.com/cure53/Flashbang.

[CW13] Erika Chin and David Wagner. Bifocals: Analyzing WebView vulnerabilities in android applications. In Yongdae Kim, Heejo Lee, and Adrian Perrig, editors, *Information Security Applications - 14th International Workshop, WISA 2013, Jeju Island, Korea, August 19-21, 2013, Revised Selected Papers*, volume 8267 of *Lecture Notes in Computer Science*, pages 138–159. Springer, 2013.

[DCJ+13] Adam Doupé, Weidong Cui, Mariusz H. Jakubowski, Marcus Peinado, Christopher Kruegel, and Giovanni Vigna. deDacota: toward preventing server-side XSS via automatic code and data separation. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 1205–1216, 2013.

[Dec10] Maarten Decat. Ondersteuning voor veilige Web Mashups. Master's thesis, Katholieke Universiteit Leuven, 2010.

[DGDNP12] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. FlowFox: a web browser with flexible and precise information flow control. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 748–759. ACM, 2012.

[DP10] Dominique Devriese and Frank Piessens. Noninterference through Secure Multi-execution. *2010 IEEE Symposium on Security and Privacy*, pages 109–124, 2010.

[DR14] Philippe De Ryck. *Client-side Web Security: Mitigating Threats against Web Sessions*. PhD thesis, December 2014.

[DRDH+10] Philippe De Ryck, Lieven Desmet, Thomas Heyman, Frank Piessens, and Wouter Joosen. CsFire: Transparent client-side mitigation of malicious cross-domain requests. In *Lecture Notes in Computer Science*, volume 5965, pages 18–34. Springer Berlin / Heidelberg, February 2010.

[DRDJP11] Philippe De Ryck, Lieven Desmet, Wouter Joosen, and Frank Piessens. Automatic and Precise Client-Side Protection against CSRF Attacks. In Vijay Atluri and Claudia Diaz, editors, *Computer Security - ESORICS 2011*, volume 6879 of *Lecture Notes in Computer Science*, pages 100–116. Springer Berlin / Heidelberg, 2011.

[DRDPP11] Philippe De Ryck, Lieven Desmet, Pieter Philippaerts, and Frank Piessens. A security analysis of next generation web standards. Technical report, G. Hogben and M. Dekker (Eds.), European Network and Information Security Agency (ENISA), July 2011.

[DRND+12] Philippe De Ryck, Nick Nikiforakis, Lieven Desmet, Frank Piessens, and Wouter Joosen. Serene: Self-reliant client-side protection against session fixation. In *12th IFIP International Conference on Distributed Applications and Interoperable Systems*, pages 59–72. Springer-Verlag, June 2012.

[DTLJ11] Xinshu Dong, Minh Tran, Zhenkai Liang, and Xuxian Jiang. AdSentry: Comprehensive and Flexible Confinement of JavaScript-based Advertisements. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 297–306, New York, NY, USA, 2011. ACM.

[Eck10] Peter Eckersley. How unique is your web browser? In Mikhail J. Atallah and Nicholas J. Hopper, editors, *Privacy Enhancing Technologies, 10th International Symposium, PETS 2010, Berlin, Germany, July 21-23, 2010. Proceedings*, volume 6205 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2010.

[ECM] ECMAScript. Harmony Direct Proxies. http://wiki.ecmascript.org/doku.php?id=harmony:direct_proxies.

[Eri] Eric Sink. Memoirs From the Browser Wars. http://ericsink.com/Browser_Wars.html.

[EWKK09] Manuel Egele, Peter Wurzinger, Christopher Kruegel, and Engin Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In Ulrich Flegel and Danilo Bruschi, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment, 6th International Conference, DIMVA 2009, Como, Italy, July 9-10, 2009. Proceedings*, volume 5587 of *Lecture Notes in Computer Science*, pages 88–106. Springer, 2009.

[FCKV09] Sean Ford, Marco Cova, Christopher Kruegel, and Giovanni Vigna. Analyzing and detecting malicious flash advertisements. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, pages 363–372, Washington, DC, USA, 2009. IEEE Computer Society.

[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.

[fla] PC Penetration | Statistics | Adobe Flash Platform runtimes. http://www.adobe.com/products/flashplatformruntimes/statistics.html.

[FWB10] Matthew Finifter, Joel Weinberger, and Adam Barth. Preventing Capability Leaks in Secure JavaScript Subsets. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2010, San Diego, California, USA, 28th February - 3rd March 2010*. The Internet Society, 2010.

[gal] Galeon. http://galeon.sourceforge.net/.

[GFLS11] Arjun Guha, Matthew Fredrikson, Benjamin Livshits, and Nikhil Swamy. Verified security for browser extensions. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 115–130, Washington, DC, USA, 2011. IEEE Computer Society.

[GG07] Alan Grosskurth and Michael W. Godfrey. A case study in architectural analysis: The evolution of the modern web browser. EMSE, 2007.

[GL09] Salvatore Guarnieri and V. Benjamin Livshits. GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In Fabian Monrose, editor, *18th USENIX Security Symposium, Montreal, Canada, August 10-14, 2009, Proceedings*, pages 151–168. USENIX Association, 2009.

[GNO] GNOME. Gjs: JavaScript Bindings for GNOME. https://wiki.gnome.org/action/show/Projects/Gjs?action=show&redirect=Gjs.

[Gol06] Dieter Gollmann. Why trust is bad for security. *Electronic Notes in Theoretical Computer Science*, 157(3):3–9, 2006.

[Gooa] Google. Chrome extensions: AdBlock. https://chrome.google.com/webstore/detail/adblock/gighmmpiobklfepjocnamgkkbiglidom.

[Goob] Google. Google Latitude. https://www.google.com/latitude/.

[Gooc] Google. Hangouts. https://www.google.com/tools/dlpage/hangoutplugin.

[Good] Google. Inside Search: Crawling & Indexing. http://www.google.com/intl/en/insidesearch/howsearchworks/crawling-indexing.html.

[Gooe] Google. V8 JavaScript Engine. https://code.google.com/p/v8/.

[Goof] Google Chrome Developers. Native Client. https://developer.chrome.com/native-client.

[gre] Greasemonkey. https://addons.mozilla.org/en-US/firefox/addon/greasemonkey/.

[GSK10] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The Essence of JavaScript. In Theo D'Hondt, editor, *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, volume 6183 of *Lecture Notes in Computer Science*, pages 126–150. Springer, 2010.

[Hei11] Mario Heiderich. Locking the Throne Room - How ES5+ will change XSS and Client Side Security. http://www.slideshare.net/x00mario/locking-the-throneroom-20, November 2011.

[Hew] Hewlett-Packard Development Company. SWFScan. http://h30499.www3.hp.com/t5/Following-the-White-Rabbit/SWFScan-FREE-Flash-decompiler/bc-p/5442703?jumpid=reg_r1002_usen.

[HFH11] Mario Heiderich, Tilman Frosch, and Thorsten Holz. IceShield: Detection and Mitigation of Malicious Websites with a Frozen DOM. In Robin Sommer, Davide Balzarotti, and Gregor Maier,

editors, *Recent Advances in Intrusion Detection*, volume 6961 of *Lecture Notes in Computer Science*, pages 281–300. Springer Berlin Heidelberg, 2011.

[HH10] I. Hickson and D. Hyatt. HTML 5 Working Draft - The sandbox Attribute. http://www.w3.org/TR/html5/the-iframe-element.html#attr-iframe-sandbox, June 2010.

[HNL14] Mitchell Hayenga, Vignyan Reddy Kothinti Naresh, and Mikko H. Lipasti. Revolver: Processor architecture for power efficient loop execution. In *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014*, pages 591–602. IEEE Computer Society, 2014.

[htm] HTML5. http://dev.w3.org/html5/spec/Overview.html.

[IETa] IETF. RFC 1866 - Hypertext Markup Language - 2.0. http://tools.ietf.org/html/rfc1866.

[IETb] IETF. RFC 2616 - Hypertext Transfer Protocol – HTTP/1.1. http://tools.ietf.org/html/rfc2616.

[IETc] IETF. RFC 3986 - Uniform Resource Identifier (URI): Generic Syntax. http://tools.ietf.org/html/rfc3986.

[IETd] IETF. RFC 6265 - HTTP State Management Mechanism. http://tools.ietf.org/html/rfc6265.

[int] Internet Archive: Wayback Machine. http://archive.org/web/.

[Inv] Involver. Tweets To Pages. http://www.facebook.com/TweetsApp.

[IW12] Lon Ingram and Michael Walfish. Treehouse: JavaScript Sandboxes to Help Web Developers Help Themselves. In Gernot Heiser and Wilson C. Hsieh, editors, *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, pages 153–164. USENIX Association, 2012.

[Jac] Jacaranda. Jacaranda. http://jacaranda.org.

[Jag09] Prajakta Jagdale. Blinded by flash: Widespread security risks flash developers don't see. In *BlackHat DC*, 2009.

[JDRC10] Karthick Jayaraman, Wenliang Du, Balamurugan Rajagopalan, and Steve J. Chapin. ESCUDO: A fine-grained protection model for web browsers. In *2010 International Conference on Distributed Computing Systems, ICDCS 2010, Genova, Italy, June 21-25, 2010*, pages 231–240. IEEE Computer Society, 2010.

[JFJ08] Markus Jakobsson, Peter Finn, and Nathaniel A. Johnson. Why and how to perform fraud experiments. *IEEE Security & Privacy*, 6(2):66–68, 2008.

[Jk] JoMo-kun. m0j0.j0j0 Guide to IIS Hacking. http://www.foofus.net/~jmk/iis.html.

[JKK06] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 258–263, 2006.

[Joh] John Resig. Pure JavaScript HTML Parser. http://ejohn.org/blog/pure-javascript-html-parser/.

[jQu] jQuery. Update on jQuery.com Compromises. http://blog.jquery.com/2014/09/24/update-on-jquery-com-compromises/.

[JR06] Markus Jakobsson and Jacob Ratkiewicz. Designing ethical phishing experiments: a study of (ROT13) rOnl query features. In Les Carr, David De Roure, Arun Iyengar, Carole A. Goble, and Michael Dahlin, editors, *Proceedings of the 15th international conference on World Wide Web, WWW 2006, Edinburgh, Scotland, UK, May 23-26, 2006*, pages 513–522. ACM, 2006.

[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.

[jsla] JSLint,The JavaScript Code Quality Tool. http://www.jslint.com/.

[JSLb] JSLint Error Explanations. Implied eval is evil. Pass a function instead of a string. http://jslinterrors.com/implied-eval-is-evil-pass-a-function-instead-of-a-string.

[JVSS11] Dongseok Jang, Aishwarya Venkataraman, G. Michael Swaka, and Hovav Shacham. Analyzing the Cross-domain Policies of Flash

Applications. In *Proceedings of the 5th Workshop on Web 2.0 Security and Privacy (W2SP)*, 2011.

[KAPM11] Georgios Kontaxis, Demetres Antoniades, Iasonas Polakis, and Evangelos P. Markatos. An empirical study on the security of cross-domain policies in rich internet applications. In Engin Kirda and Steven Hand, editors, *Proceedings of the Fourth European Workshop on System Security, EUROSEC'11, April 10, 2011, Salzburg, Austria*, page 7. ACM, 2011.

[KKVJ06] Engin Kirda, Christopher Krügel, Giovanni Vigna, and Nenad Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In Hisham Haddad, editor, *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC), Dijon, France, April 23-27, 2006*, pages 330–337. ACM, 2006.

[Kle05] Amit Klein. DOM Based Cross Site Scripting or XSS of the Third Kind. http://www.webappsec.org/projects/articles/071105.shtml, April 2005.

[KLZS12] Clemens Kolbitsch, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. Rozzle: De-cloaking internet malware. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 443–457, Washington, DC, USA, 2012. IEEE Computer Society.

[Kog] Igor Kogan. no|wrap.be - flare. http://www.nowrap.de/flare.html.

[Kri] Kris Zyp. Secure Mashups with dojox.secure. http://www.sitepen.com/blog/2008/08/01/secure-mashups-with-dojoxsecure/.

[LD11] Tongbo Luo and Wenliang Du. Contego: Capability-based access control for web browsers - (short paper). In Jonathan M. McCune, Boris Balacheff, Adrian Perrig, Ahmad-Reza Sadeghi, Angela Sasse, and Yolanta Beres, editors, *Trust and Trustworthy Computing - 4th International Conference, TRUST 2011, Pittsburgh, PA, USA, June 22-24, 2011. Proceedings*, volume 6740 of *Lecture Notes in Computer Science*, pages 231–238. Springer, 2011.

[LGV10] Mike Ter Louw, Karthik Thotta Ganesh, and V. N. Venkatakrishnan. Adjail: Practical enforcement of confidentiality and integrity policies on web advertisements. In *19th USENIX Security Symposium, Washington, DC, USA, August 11-13, 2010, Proceedings*, pages 371–388. USENIX Association, 2010.

[LJT11] Sebastian Lekies, Martin Johns, and Walter Tighzert. The state of the cross-domain nation. In *Proceedings of the 5th Workshop on Web 2.0 Security and Privacy (W2SP)*, 2011.

[Lou] Louis J. Montulli II. the origin of the <blink> tag - www. http://www.montulli.org/theoriginofthe%3Cblink%3Etag.

[LSJ13] Sebastian Lekies, Ben Stock, and Martin Johns. 25 million flows later: large-scale detection of DOM-based XSS. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 1193–1204. ACM, 2013.

[LV09] Mike Ter Louw and V. N. Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers, 2009.

[LZW10] Zhou Li, Kehuan Zhang, and XiaoFeng Wang. Mash-IF: Practical information-flow control within client-side mashups. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pages 251 –260, June 28 2010 – July 1 2010 2010.

[LZYC12] Lei Liu, Xinwen Zhang, Guanhua Yan, and Songqing Chen. Chrome extensions: Threat analysis and countermeasures. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*. The Internet Society, 2012.

[Mao11] Giorgio Maone. NoScript 2.0.9.9. http://noscript.net/, 2011.

[MAS10] Jonas Magazinius, Aslan Askarov, and Andrei Sabelfeld. A lattice-based approach to mashup security. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 15–23, New York, NY, USA, 2010. ACM.

[Max] Maxthon. Maxthon Cloud Browser. http://www.maxthon.com/.

[MF12] James Mickens and Matthew Finifter. Jigsaw: Efficient, low-effort mashup isolation. In *Presented as part of the 3rd USENIX Conference on Web Application Development (WebApps 12)*, pages 13–25, Boston, MA, 2012. USENIX.

[MFM10] Leo A. Meyerovich, Adrienne Porter Felt, and Mark S. Miller. Object views: fine-grained sharing in browsers, 2010.

[Mica] Microsoft. HTML Tag Methods (JavaScript). http://msdn.microsoft.com/en-us/library/ie/ff806183(v=vs.94).aspx.

[Micb] Microsoft. IE8 Security Part IV: The XSS Filter. http://blogs.msdn.com/b/ie/archive/2008/07/02/ie8-security-part-iv-the-xss-filter.aspx.

[Micc] Microsoft. Internet Explorer Architecture. http://msdn.microsoft.com/en-us/library/aa741312(v=vs.85).aspx.

[Micd] Microsoft. marquee element | marquee object (Internet Explorer). http://msdn.microsoft.com/en-us/library/ie/ms535851(v=vs.85).aspx.

[Mice] Microsoft. Microsoft Internet Security and Acceleration (ISA) Server 2004. http://technet.microsoft.com/en-us/library/cc302436.aspx.

[Micf] Microsoft. Microsoft Security Bulletin MS04-040 - Critical. https://technet.microsoft.com/en-us/library/security/ms04-040.aspx.

[micg] Microsoft Silverlight. http://www.microsoft.com/silverlight/.

[Mich] Microsoft. Mitigating Cross-site Scripting With HTTP-only Cookies. http://msdn.microsoft.com/en-us/library/ms533046(VS.85).aspx.

[Mici] Microsoft. Security in Silverlight. http://msdn.microsoft.com/en-us/library/cc972657(v=vs.95).aspx.

[Micj] Microsoft Live Labs. Live Labs Websandbox. http://websandbox.org.

[Mih] Mihai Bazon. UglifyJS. https://github.com/mishoo/UglifyJS/.

[Mil] Mark Samuel Miller. Secure EcmaScript 5. http://code.google.com/p/es-lab/wiki/SecureEcmaScript.

[Mil06] Mark Samuel Miller. *Robust composition: towards a unified approach to access control and concurrency control*. PhD thesis, Johns Hopkins University, Baltimore, MD, USA, 2006. AAI3245526.

[Min] Miniwatts Marketing Group. Internet usage statistics. http://www.internetworldstats.com/stats.htm.

[MIT] MITRE. CWE-367: Time-of-check Time-of-use (TOCTOU) Race Condition. http://cwe.mitre.org/data/definitions/367.html.

[ML10] Leo A. Meyerovich and V. Benjamin Livshits. ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 481–496. IEEE Computer Society, 2010.

[MMT09] Sergio Maffeis, John C. Mitchell, and Ankur Taly. Isolating JavaScript with Filters, Rewriting, and Wrappers. In Michael Backes and Peng Ning, editors, *Computer Security - ESORICS 2009, 14th European Symposium on Research in Computer Security, Saint-Malo, France, September 21-23, 2009. Proceedings*, volume 5789 of *Lecture Notes in Computer Science*, pages 505–522. Springer, 2009.

[MMT10] Sergio Maffeis, John C. Mitchell, and Ankur Taly. Object capabilities and isolation of untrusted web applications. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 125–140. IEEE Computer Society, 2010.

[Moza] Mozilla. Gecko. https://developer.mozilla.org/en-US/docs/Mozilla/Gecko.

[Mozb] Mozilla. JavaScript Strict Mode Reference. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode.

[Mozc] Mozilla. Most popular Firefox extensions. https://addons.mozilla.org/en-US/firefox/extensions/?sort=users.

[Mozd] Mozilla. Mozilla MDN - <marquee>. https://developer.mozilla.org/en-US/docs/Web/HTML/Element/marquee.

[Moze] Mozilla. Shumway. http://mozilla.github.io/shumway/.

[Mozf] Mozilla. The Narcissus meta-circular JavaScript interpreter. https://github.com/mozilla/narcissus.

[Mozg] Mozilla. The “with” statement. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/with.

[Mozh] Mozilla. XSS Filter. https://wiki.mozilla.org/Security/Features/XSS_Filter.

[MPS10] Jonas Magazinius, Phu H. Phung, and David Sands. Safe Wrappers and Sane Policies for Self Protecting JavaScript. In Tuomas Aura, Kimmo Järvinen, and Kaisa Nyberg, editors, *Information Security Technology for Applications - 15th Nordic Conference on Secure IT Systems, NordSec 2010, Espoo, Finland, October 27-29, 2010, Revised Selected Papers*, volume 7127 of *Lecture Notes in Computer Science*, pages 239–255. Springer, 2010.

[MSL+08] Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja - safe active content in sanitized JavaScript. Technical report, Google Inc., June 2008.

[MT09] Sergio Maffeis and Ankur Taly. Language-Based Isolation of Untrusted JavaScript. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF 2009, Port Jefferson, New York, USA, July 8-10, 2009*, pages 77–91. IEEE Computer Society, 2009.

[NBVA+11] Nick Nikiforakis, Marco Balduzzi, Steven Van Acker, Wouter Joosen, and Davide Balzarotti. Exposing the lack of privacy in file hosting services. In *Proceedings of the 4th USENIX conference on Large-scale exploits and emergent threats*, LEET'11, pages 8–8, Berkeley, CA, USA, 2011. USENIX Association.

[NCS] NCSA. NSCA Mosaic. http://www.ncsa.illinois.edu/enabling/mosaic.

[nes] EU-FP7 NESSoS. http://www.nessos-project.eu/.

[neta] Netscape 2.0 reviewed. http://www.antipope.org/charlie/old/journo/netscape.html.

[netb] Netscape History - Web Browser, Marc Andreessen, Jim Clark, Mosaic. http://www.livinginternet.com/w/wi_netscape.htm.

[Netc] Netcraft. April 2014 Web Server Survey. http://news.netcraft.com/archives/2014/04/02/april-2014-web-server-survey.html.

[NIK+12] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You are what you include: large-scale evaluation of remote JavaScript inclusions. In Ting Yu,

George Danezis, and Virgil D. Gligor, editors, *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 736–747. ACM, 2012.

[NMY+11] Nick Nikiforakis, Wannes Meert, Yves Younan, Martin Johns, and Wouter Joosen. SessionShield: Lightweight protection against session hijacking. In Úlfar Erlingsson, Roel Wieringa, and Nicola Zannone, editors, *Engineering Secure Software and Systems - Third International Symposium, ESSoS 2011, Madrid, Spain, February 9-10, 2011. Proceedings*, volume 6542 of *Lecture Notes in Computer Science*, pages 87–100. Springer, 2011.

[nod] node.js. http://nodejs.org/.

[NoM] NoMoreSleep. jquery-geolocation. http://code.google.com/p/jquery-geolocation/.

[Ope] Opera. Opera Browser. http://www.opera.com.

[OWAa] OWASP. Cross-site Scripting (XSS). https://www.owasp.org/index.php/Cross-site_Scripting_(XSS).

[OWAb] OWASP. Data Validation. https://www.owasp.org/index.php/Data_Validation.

[owac] OWASP Top 10 Web Application Security Risks. http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project.

[OWAd] OWASP. Session Hijacking Attack. https://www.owasp.org/index.php/Session_hijacking_attack.

[par] SpiderMonkey - Parser API. https://developer.mozilla.org/en-US/docs/SpiderMonkey/Parser_API.

[PD12] Phu H. Phung and Lieven Desmet. A two-tier sandbox architecture for untrusted JavaScript. In *JSTools '12 Proceedings of the Workshop on JavaScript Tools, Beijing,13 June, 2012*, pages 1–10, 2012.

[PDL+11] Kailas Patil, Xinshu Dong, Xiaolei Li, Zhenkai Liang, and Xuxian Jiang. Towards Fine-Grained Access Control in JavaScript Contexts. In *2011 International Conference on Distributed Computing Systems, ICDCS 2011, Minneapolis, Minnesota, USA, June 20-24, 2011*, pages 720–729. IEEE Computer Society, 2011.

[PEGK11] Joe Gibbs Politz, Spiridon Aristides Eliopoulos, Arjun Guha, and Shriram Krishnamurthi. ADsafety: Type-Based Verification of JavaScript Sandboxing. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*. USENIX Association, 2011.

[PMS+14] P. Phung, M. Monshizadeh, M. Sridhar, K. Hamlen, and V. Venkatakrishnan. Between Worlds: Securing Mixed JavaScript/ActionScript Multi-party Web Content. *Dependable and Secure Computing, IEEE Transactions on*, PP(99):1–1, 2014.

[Pro] Programmable Web. Keeping you up to date with APIs, mashups and the Web as platform. http://www.programmableweb.com/.

[PSC09] Phu H. Phung, David Sands, and Andrey Chudnov. Lightweight self-protecting JavaScript. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, ASIACCS '09, pages 47–60, New York, NY, USA, 2009. ACM.

[qui] QuirksMode - for all your browser quirks. http://www.quirksmode.org/.

[Raf14] Willard Rafnsson. *Securing Interactive Systems*. PhD thesis, Chalmers University of Technology, 2014.

[RDD+10] Philippe De Ryck, Maarten Decat, Lieven Desmet, Frank Piessens, and Wouter Joosen. Security of web mashups: A survey. In Tuomas Aura, Kimmo Järvinen, and Kaisa Nyberg, editors, *Information Security Technology for Applications - 15th Nordic Conference on Secure IT Systems, NordSec 2010, Espoo, Finland, October 27-29, 2010, Revised Selected Papers*, volume 7127 of *Lecture Notes in Computer Science*, pages 223–238. Springer, 2010.

[RDW+06] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. BrowserShield: vulnerability-driven filtering of dynamic HTML. In *OSDI '06: Proceedings of the 7th symposium on Operating Systems Design and Implementation*, pages 61–74, Berkeley, CA, USA, 2006. USENIX Association.

[RHBV11] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The Eval That Men Do - A Large-Scale Study of the Use of Eval in JavaScript Applications. In Mira Mezini, editor, *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings*, volume 6813 of *Lecture Notes in Computer Science*, pages 52–78. Springer, 2011.

[ria] Rich Internet Application (RIA) Market Share. http://www.statowl.com/custom_ria_market_penetration.php.

[RKD10] Konrad Rieck, Tammo Krueger, and Andreas Dewald. Cujo: Efficient detection and prevention of drive-by-download attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 31–39, New York, NY, USA, 2010. ACM.

[RLZ09] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 169–186, Berkeley, CA, USA, 2009. USENIX Association.

[Rom03] Donna M Romano. *The nature of trust: conceptual and operational clarification*. PhD thesis, Louisiana State University, 2003.

[Rud] Jesse Ruderman. Configurable Security Policies. http://www.mozilla.org/projects/security/components/ConfigPolicy.html.

[Sam11] Justin Samuel. RequestPolicy 0.5.20. http://www.requestpolicy.com, 2011.

[SAN] SANS. CWE/SANS TOP 25 Most Dangerous Software Errors . http://www.sans.org/top25-software-errors/.

[SAN09] SANS Institute. SANS: Top Cyber Security Risks. http://www.sans.org/top-cyber-security-risks/, 2009.

[Shi] Chris Shiflett. Cross-Site Request Forgeries. http://shiflett.org/articles/cross-site-request-forgeries.

[SM03] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[SML11] Prateek Saxena, David Molnar, and Benjamin Livshits. ScriptGard: automatic context-sensitive sanitization for large-scale legacy web applications. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 601–614. ACM, 2011.

[sod] IWT-SBO SEC SODA. https://distrinet.cs.kuleuven.be/projects/secsoda/.

[sot] Sothink SWF decompiler. http://www.sothink.com/product/flashdecompiler/.

[spia] IWT-SBO SPION. http://www.spion.me/.

[spib] Mozilla SpiderMonkey. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey.

[SSM10] Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the web with Content Security Policy. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 921–930, New York, NY, USA, 2010. ACM.

[Ste] Stefano Di Paola. SWFIntruder. http://code.google.com/p/swfintruder/.

[Sto95] Clifford Stoll. The internet? bah! http://www.thedailybeast.com/newsweek/1995/02/26/the-internet-bah.html, 1995.

[str] EU-FP7 STREWS. https://www.strews.eu/.

[SXS11] Fangqi Sun, Liang Xu, and Zhendong Su. Static detection of access control vulnerabilities in web applications. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*. USENIX Association, 2011.

[SYM+14] Deian Stefan, Edward Z. Yang, Petr Marchenko, Alejandro Russo, David Herman, Brad Karp, and David Mazières. Protecting Users by Confining JavaScript with COWL. In Jason Flinn and Hank Levy, editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, pages 131–146. USENIX Association, 2014.

[TEM+11] Ankur Taly, Úlfar Erlingsson, John C. Mitchell, Mark S. Miller, and Jasvir Nagra. Automated Analysis of Security-Critical JavaScript APIs. In *IEEE Symposium on Security and Privacy*, pages 363–378, 2011.

[The] The FaceBook Team. FBJS. http://wiki.developers.facebook.com/index.php/FBJS.

[Tima] Tim Berners-Lee. WorldWideWeb, the first Web client. http://www.w3.org/People/Berners-Lee/WorldWideWeb.html.

[Timb] Timothy Berners-Lee. Sir Timothy Berners-Lee Interview – Academy of Achievement. http://www.achievement.org/autodoc/page/ber1int-1.

[TLPKV13] Mike Ter Louw, Phu H. Phung, Rohini Krishnamurti, and VenkatN. Venkatakrishnan. SafeScript: JavaScript Transformation for Policy Enforcement. In Hanne Riis Nielson and Dieter Gollmann, editors, *Secure IT Systems*, volume 8208 of *Lecture Notes in Computer Science*, pages 67–83. Springer Berlin Heidelberg, 2013.

[top] Mozilla add-ons - featured extensions. https://addons.mozilla.org/en-US/firefox/extensions/.

[tre] Trend Micro Site Safety Center. http://global.sitesafety.trendmicro.com/.

[tru] iMinds-ICON TRUBLISS. https://distrinet.cs.kuleuven.be/research/projects/TRU-BLISS.

[Twi] Twitter. How to embed Twitter timelines on your website. https://blog.twitter.com/2012/embedded-timelines-howto.

[use] Userscripts.org. http://userscripts.org/.

[uso] Userscripts-mirror.org: Power-ups for your browser. http://userscripts-mirror.org/.

[Vanar] Steven Van Acker. Frameworks for, and lessons from large-scale experimentation. Technical report, to appear.

[VCM12] Tom Van Cutsem and Mark S. Miller. On the Design of the ECMAScript Reflection API. Technical Report VUB-SOFT-TR-12-03, Department of Computer Science, Vrije Universiteit Brussel, February 2012.

[VGC09] 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 Network and Distributed System Security Symposium (NDSS)*. The Internet Society, February 2009.

[vGCN+14] Tom van Goethem, Ping Chen, Nick Nikiforakis, Lieven Desmet, and Wouter Joosen. Large-scale security analysis of the web: Challenges and findings. In Thorsten Holz and Sotiris Ioannidis, editors, *Trust and Trustworthy Computing*, volume 8564 of *Lecture Notes in Computer Science*, pages 110–126. Springer International Publishing, 2014.

[VND+14] Steven Van Acker, Nick Nikiforakis, Lieven Desmet, Frank Piessens, and Wouter Joosen. Monkey-in-the-browser: Malware and

vulnerabilities in augmented browsing script markets – extended version. Technical report, March 2014.

[VNJ+07] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Krügel, and Giovanni Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. 2007.

[W3Ca] W3C. About W3C Standards. http://www.w3.org/standards/about.html.

[W3Cb] W3C. Content Security Policy. https://w3c.github.io/webappsec/specs/content-security-policy/.

[W3Cc] W3C. Current Members - W3C. http://www.w3.org/Consortium/Member/List.

[W3Cd] W3C. Document Object Model (DOM) Technical Reports. http://www.w3.org/DOM/DOMTR.

[W3Ce] W3C. Same Origin Policy - Web Security. http://www.w3.org/Security/wiki/Same_Origin_Policy.

[W3Cf] W3C. The history of the Web - W3C Wiki. http://www.w3.org/wiki/The_history_of_the_Web.

[W3Cg] W3C. The SCRIPT element: Specifying the scripting language. http://www.w3.org/TR/html4/interact/scripts.html#h-18.2.2.

[W3Ch] W3C. W3C Standards and drafts - Cross-Origin Resource Sharing. http://www.w3.org/TR/cors/.

[W3Ci] W3C. W3C Standards and drafts - JavaScript APIs. http://www.w3.org/TR/#tr_JavaScript_APIs.

[W3Cj] W3C. W3C Standards and drafts - Uniform Messaging Policy, Level One. http://www.w3.org/TR/UMP/.

[W3Ck] W3C. XML Path Language (XPath) 2.0. http://www.w3.org/TR/xpath20/.

[WBJ+06] Yi-Min Wang, Doug Beck, Xuxian Jiang, Roussi Roussev, Chad Verbowski, Shuo Chen, and Samuel T. King. Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites That Exploit Browser Vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2006, San Diego, California, USA*. The Internet Society, 2006.

[weba] EU-FP7 WebSand. https://www.websand.eu/.

[Webb] Webkit Blog - David Carson. Android uses WebKit. https://www.webkit.org/blog/142/android-uses-webkit/.

[WHA] WHATWG. HTML Living Standard - Timers. https://html.spec.whatwg.org/multipage/webappapis.html#timers.

[Wil] Willem De Groef. ConScript For Firefox. http://cqrit.be/conscript/.

[WLR14] Michael Weissbacher, Tobias Lauinger, and William K. Robertson. Why is CSP failing? trends and challenges in CSP adoption. In Angelos Stavrou, Herbert Bos, and Georgios Portokalidis, editors, *Research in Attacks, Intrusions and Defenses - 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings*, volume 8688 of *Lecture Notes in Computer Science*, pages 212–233. Springer, 2014.

[WS07] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 32–41, New York, NY, USA, 2007. ACM.

[XA06] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In Angelos D. Keromytis, editor, *Proceedings of the 15th USENIX Security Symposium, Vancouver, BC, Canada, July 31 - August 4, 2006*. USENIX Association, 2006.

[xss] The Cross-site Scripting FAQ. http://www.cgisecurity.com/xss-faq.html.

[Yah] Yahoo! Developer Network. JavaScript: Use a Web Proxy for Cross-Domain XMLHttpRequest Calls. http://developer.yahoo.com/javascript/howto-proxy.html.

[YCIS07] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. JavaScript Instrumentation for Browser Security. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 237–249, New York, NY, USA, 2007. ACM.

[YW09] Chuan Yue and Haining Wang. Characterizing insecure JavaScript practices on the web. In Juan Quemada, Gonzalo León, Yoëlle S.

Maarek, and Wolfgang Nejdl, editors, *Proceedings of the 18th International Conference on World Wide Web, WWW 2009, Madrid, Spain, April 20-24, 2009*, pages 961–970. ACM, 2009.

[Zal10] M. Zalewski. Browser security handbook. http://code.google.com/p/browsersec/wiki/Main, 2010.

[ZDN] ZDNet. ZDNet Products: A brief history of cyberspace. http://www.cs.duke.edu/courses/spring01/cps049s/class/html/mp.history.html.

List of Publications

Papers at international scientific conferences and symposia, published in full in proceedings

- Van Acker, S., Hausknecht, D., Joosen, W., Sabelfeld, A. (2015). Password Meters and Generators on the Web: From Large-Scale Empirical Study to Getting It Right. Proceedings of the Fifth ACM Conference on Data and Application Security and Privacy (CODASPY 2015). San Antonio, TX, USA, 2-4 March 2015, Accepted.
- Van Acker, S., Nikiforakis, N., Desmet, L., Piessens, F., Joosen, W. (2014). Monkey-in-the-browser: Malware and vulnerabilities in augmented browsing script markets. ASIACCS. Kyoto, Japan, 2-4 June 2014.
- Nikiforakis, N., Van Acker, S., Meert, W., Desmet, L., Piessens, F., Joosen, W. (2013). Bitsquatting: Exploiting bit-flips for fun, or profit?. Proceedings of the 22nd International World Wide Web Conference. World Wide Web Conference (WWW). Rio De Janeiro, Brazil, 13-17 May 2013 (pp. 989-998).
- Nikiforakis, N., Invernizzi, L., Kapravelos, A., Van Acker, S., Joosen, W., Kruegel, C., Piessens, F., Vigna, G. (2012). You are what you include: Large-scale evaluation of remote JavaScript inclusions. Proceedings of the 19th ACM conference on Computer and Communications Security (CCS 2012). Raleigh, NC, USA, 16-18 October 2012 (pp. 736-747).
- Agten, P., Van Acker, S., Brondsema, Y., Phung, P., Desmet, L., Piessens, F. (2012). JSand: Complete client-side sandboxing of third-party JavaScript without browser modifications. Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC 2012). Orlando, Florida, USA, 3-7 December 2012 (pp. 1-10).

- Van Acker, S., Nikiforakis, N., Desmet, L., Joosen, W., Piessens, F. (2012). FlashOver: Automated discovery of cross-site scripting vulnerabilities in rich internet applications. ASIACCS. Seoul, 2-4 May 2012.
- Nikiforakis, N., Van Acker, S., Piessens, F., Joosen, W. (2012). Exploring the ecosystem of Referrer-Anonymizing Services. Privacy Enhancing Technologies Symposium (PETS 2012). Vigo, Spain, 11-13 July 2012.
- Van Acker, S., De Ryck, P., Desmet, L., Piessens, F., Joosen, W. (2011). WebJail: Least-privilege integration of third-party components in web mashups. Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC 2011). Orlando, Florida, USA, 5-9 December 2011 (pp. 307-316).
- Nikiforakis, N., Balduzzi, M., Van Acker, S., Joosen, W., Balzarotti, D. (2011). Exposing the lack of privacy in file hosting services. Usenix Workshop on Large-Scale Exploits and Emergent Threats. Boston, US, 29 March 2011.
- Van Acker, S., Nikiforakis, N., Philippaerts, P., Younan, Y., Piessens, F. (2010). ValueGuard: Protection of native applications against data-only buffer overflows. Lecture Notes in Computer Science: Vol. 6503. International Conference on Information Systems Security (ICISS 2010). Gandhinagar, India, 15-19 December 2010 (pp. 156-170).

Internal report

- Van Acker, S., Nikiforakis, N., Desmet, L., Piessens, F., Joosen, W. (2014). Monkey-in-the-browser: Malware and vulnerabilities in augmented browsing script markets – extended version. CW Reports, CW657, 11 pp. Leuven, Belgium: Department of Computer Science, KU Leuven.

Mischief Managed.

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
SCIENTIFIC COMPUTING GROUP
Celestijnenlaan 200A box 2402
B-3001 Heverlee
steven.vanacker@cs.kuleuven.be
http://www.cs.kuleuven.be