Citation for published version:
Viopoulou, E 2014, *MSc Dissertation: An investigation of JavaScript isolation mechanisms: Sandboxing implementations*. Department of Computer Science Technical Report Series, no. CSBU-2014-03, Department of Computer Science, Bath U.K.

Publication date:
2014

Document Version
Early version, also known as pre-print

Link to publication

Publisher Rights
CC BY-NC-ND

University of Bath

Alternative formats
If you require this document in an alternative format, please contact:
openaccess@bath.ac.uk

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 01. Dec. 2021

Department of Computer Science

Technical Report

MSc Dissertation: An investigation of JavaScript isolation mechanisms: Sandboxing implementations

Efthymia Viopoulou

Technical Report 2014-03 October 2014

ISSN 1740-9497

Copyright ©October 2014 by the authors.

Contact Address:
Department of Computer Science
University of Bath
Bath, BA2 7AY
United Kingdom
URL: http://www.cs.bath.ac.uk

ISSN 1740-9497

An investigation of JavaScript isolation mechanisms

Sandboxing implementations

Efthymia Viopoulou

Master of Science in Internet Systems and Security
University of Bath
September 2014

Abstract

Web developers have been relying more and more on the features of JavaScript for deploying a range of applications, from online banking and email services to digital media delivery and gaming, rendering it the assembly language of the Internet. Since it allows external scripts from untrusted third-parties to access the application's content, it has become an attractive target for cyber attackers. This untested and untrusted by the application's true author code cannot only damage the content of the application it runs within, but also obtain access and modify other applications running on the same browser, or even the host system.

In this paper, we compare implementations that operate as Sandboxes in order to isolate JavaScript from untrusted third-parties, so that they have limited privileges over the application. We use specific metrics to compare them, and afterwards we evaluate those results after testing them on an x64 machine.

Contents

1 Introduction **1**

2 Literature Review **8**

2.1 Language-based approaches 8
2.1.1 Static Checks 9
2.1.2 Program Instrumentation 12
2.1.3 JavaScript type-systems 17
2.2 Auditing mechanisms 18
2.2.1 Intrusion Detection System 18
2.2.2 SABRE 19
2.3 Interpreters 19
2.3.1 Safe interpreters 20
2.3.2 js.js 20
2.4 Web workers 20
2.4.1 Treehouse 21
2.4.2 JSandbox 22
2.4.3 Bawks 22
2.5 Browser-based approaches 22
2.5.1 Browser Operating System 22
2.5.2 Frame isolation 26
2.6 Others 28

3 JavaScript Overview **32**

3.1 Previous uses 32
3.2 Features-exploits-dynamism 34

4 Overview of the various solutions **36**

4.1 Pre-requisites 36

4.2 TreeHouse . . . 37
4.2.1 Initial Experiments . . . 37
4.2.2 After contacting the author . . . 40
4.3 JavaScript in JavaScript . . . 43

5 Experimental results **49**
5.1 Evaluation . . . 49
5.1.1 TreeHouse . . . 49
5.1.2 JS in JS . . . 55
5.2 Comparisons . . . 62
5.2.1 Head-to-head comparisons . . . 62
5.2.2 Comparisons to prior work . . . 64

6 Evaluation **66**

7 Conclusion **71**
7.1 Summary . . . 71
7.2 Future Work . . . 72

A Project Diary **74**

List of Figures

4.1 Our home page 36
4.2 TreeHouse: DOM manipulation (innerHTML) 42

5.1 Page load latency on Chrome 50
5.2 Page load latency on Chrome in milliseconds 60

List of Tables

5.1 DOMTRIS page load latency with TreeHouse in milliseconds 51
5.2 DOMTRIS page load latency without TreeHouse in milliseconds 53
5.3 Mean values in milliseconds 53
5.4 NewRuntime time load in milliseconds 57
5.5 NewContext time load in milliseconds 57
5.6 GlobalClassInit time load in milliseconds 58
5.7 StandardClassesInit time load in milliseconds 58
5.8 Execute 1+1 time load in milliseconds 59
5.9 DestroyContext time load in milliseconds 59
5.10 DestroyRuntime time load in milliseconds 60
5.11 Mean time load in milliseconds 61
5.12 Mean time load of start up, simple execution, and shut down routines in milliseconds 61
5.13 TreeHouse and js.js design differences 63
5.14 Author's 64
5.15 Ours 64
5.16 TreeHouse: Comparison of results 64
5.17 Author's 65
5.18 Ours 65
5.19 JS in JS: Comparison of results 65

6.1 Hours spent on the whole project based on hour diary 68

Acknowledgements

First and foremost I would like to thank my supervisor, Dr James Harold Davenport, for his abundant help and guidance throughout this project. I also take this opportunity to express my gratitude to Mr Lon Ingram, one of the authors of one of the implementations that we evaluated through this project, for the technical assistance he provided and his willingness to help. Mr Fabio Nemetz, the Project Coordinator, for the interest and the guidance that he provided. Finally, I would like to thank my family for all the support and encouragement they showed me during the project.

Chapter 1

Introduction

JavaScript [Fla02] is a scripting language developed by Netscape and designed for enhancing Web pages, in order to make them more responsive, engaging and interactive, by defining event handlers when a specific event occurs. Its use in Web browsers allows executable content to be distributed over the Internet in the form of JavaScript scripts, but its role is not restricted to that; JavaScript can also be embedded within any application. It allows control not only over the context of Web pages, but also over the browser and over the content and behavior of the HTML forms that appear in the browser. Moreover, its ability to handle cookies [Fla02] has enabled their use not only by server-side scripts, but by client-side as well; the HTTP protocol by design is a stateless protocol [GWS11], whereas web applications require stateful sessions. The use of cookies is an attempt at reintroducing the notion of state which is essential for security. Thus, with JavaScript, programs have the ability to read and write cookie values both by the server and by the client end.

However, since JavaScript is used for creating documents transmitted over the Internet, it can also be exploited to launch browser-based security attacks. Due to the complexity of the Web browser environment, and the fact that JavaScript became the standard for executable content on the Web, it became an attractive target for malicious attackers and security problems arose. Users' credentials can be stolen, or benign web applications can be fooled to deliver harmful content to the user with cross-site attacks [CHGL06, HV05, DLFMT04]. Another form of attack is the spyware infection (drive-by-download) [SGL04], where the user clicks on a hyperlink

that could cause hostile software to be downloaded and executed on the user's machine, either by using frameworks [teaa], [Teab], [jT], [Tead], [Teaf], [Teag], JavaScript libraries that abstract messy aspects of the browser's interface, or by using ad networks [IW12] that can hinder or harm the enclosing page, or by using widgets [Teae], code supplied by an off-site service to invoke that very service. Content-based attacks [CHGL06] caused by security holes of trusted plug-ins are very common, as well as hijacking browser windows [Chi06], [Pro], [saW], [Sec], caused by browser flaws. Users could also be lured into providing sensitive information to unauthorized parties with hijacking attacks [CHGL06, HV05, YCIS07, DG09], where a hostile application masquerades as another to capture information from the user. Moreover, a malicious attacker could steal sensitive information from the user such as cookies [MMT09] and saved passwords, execute arbitrary code on the host system, compromise the browser security, or even exploit the privileges that the browser affords to JavaScript [DG09] due to the fact that JavaScript extensions from untrusted third-parties may contain malicious functionality. Vulnerabilities in the browser could cause the file system on the host machine to be accessed by unauthorized parties, or system resources to be abused [Pil, Wil05], whereas session riding and self-propagating worms in Web-based email and social networking sites [Chi06], [Pro], [Pro] are both very well known attacks. All these attacks lead to the conclusion that JavaScript, thus Web applications and Web browsers [CHGL06] cannot be trusted. However, despite the threats that JavaScript code could bring to users, caused by either malicious injected scripts or poorly designed third-party code, most of them still trust it running on their browsers.

There have been many attempts to address these issues and successfully thwart these attacks, all of them trying to solve the same problem, but following different approaches. Previous work has been done with language-based approaches, auditing mechanisms, Web Workers, Interpreters, and browser-based approaches.

Static checks - a language based approach that involves code checking before it is executed

- GateKeeper [GL09], a static analyzer that protects the host from malicious widgets
- FBJS [MT09], a subset of JavaScript for the Facebook networking site in which every part of the application's code is rewritten to that safe subset
- ADsafe [Cro08], a subset of JavaScript that filters specific methods in order to isolate advertisements
- Jacaranda, Dojo Secure, and Blancura [FWB10], all used for web page advertising isolation
- Type refinement [KSW+13]

Apart from static checks, program instrumentation has been studied, where the code is checked dynamically with filters, rewriters, and wrappers, after its execution

- [MMT09] found vulnerabilities in Adsafe and BFJS, and proposes dynamic checking as a solution by combining filtering, rewriting, and wrapping [MMT09]
- Caja [MSL+08] is a mechanism that filters and rewrites the code
- WebSandbox [IM09],BrowserShield [RDW+07], and AjaxScope [KL10] offer dynamic instrumentation
- [PSC09] is another example of dynamic instrumentation
- Mutation-Event Transforms [ELX07] is a client mechanism that runs before any other script and disallows the use of certain web client features
- CoreScript [YCIS07] is a mechanism for regulating the behavior of untrusted JavaScript code by rewriting scripts

- XPCNativeWrapper [Cen], a JavaScript object for wrapping properties that should be used whenever privileged code is used to access unprivileged code
- JS_0 [AGD05] for better error detection
- [Thi05], a type system for analyzing JavaScript programs
- [CF91] combines static and dynamic typing

Auditing mechanisms have been developed like:

- the Intrusion Detection System [HV05]
- SABRE [DG09], a Security Architecture for Browser Extensions

Work on Interpreters, JavaScript engines that compile JavaScript code, includes:

- Safe Interpreters [AM98]
- js.js [TBK12], an interpreter that runs on top of JavaScript

Web Workers as separate threads to isolate JavaScript code

- TreeHouse [IW12]
- JSandbox [Gre]
- Bawks [The]

Finally, browser-based approaches that use special browser operating systems with their own security architecture include

- Tahoma [CHGL06]
- the Illinois Browser Operating System [TMK10]
- Google Chrome [RBP09]
- Internet Explorer [Zei]
- The OP browser [GTK08]
- The Atlantis browser [MD11]
- The Gazelle browser [WGM+09]

Frame isolation

- [BJM09]
- SMash
- Subspace [JW07]
- Open Mashup OS (OMOS) [ZYG08]
- AdJail [TLGV10]
- AdSentry [DTLJ11]

Others

- MashupOS [WFHJ07]
- OMash [CHC08]
- BEEP [FHEW08]
- SpoofGuard [BML+07]
- ConScript [ML10]

- BFlow [YNKM09]
- Mugshot [MEH10]

Sandboxing can also take place on hosts

- V32 [FC08]
- Ostia [GPR04]
- using software fault isolation with MisFit [SS98]
- Remote Procedure Calling [WLAG94]

It can also take place in the browser, sandboxing native code with:

- Xax [DEHL08], a browser plug-in for deploying desktop applications on the web
- NativeClient [YSD^{+}09]

A classic dilemma [PS01] is when a user wants to see the content of a received file, but is afraid of what will possibly happen to his host system in case the file contains a virus. Our project takes that dilemma to the next step, where the user is reluctant to trust any application coming from the web. On the other hand, there is the type of users [CHGL06] that assume that Web applications cannot interfere with one another or with the browser itself, which leads them to executing untrusted third party content without realizing it. The sandboxing idea is not new and tries to give a solution to the JavaScript isolation problem [MMT09], where a hosting page, P_{host}, includes content $P_1, \ldots, P_k$ from untrusted origins that will execute in the same JavaScript environment as P_{host}. This problem is difficult because of the close integration of JavaScript with complex browser applications, such as Mozilla [HV05], however, many different sandboxing implementations have been created in order to execute programs in a restricted environment;

our goal is to investigate further the sandboxing idea and discuss these different approaches that contribute to the solution of the JavaScript isolation problem. We did that by installing their code on an x64 Windows machine, evaluating them according to specific metrics, and comparing them, concluding to how effective and easy these approaches are, and how much overhead they cause. The authors' reviews but ours as well will be considered.

The rest of the paper is organized as follows. In section 2, we give the literature review, describing in more depth previous implementations that were meant to solve the JavaScript isolation problem. In section 3 we give an overview of the JavaScript language and its features. Section 4 demonstrates the steps we followed in order to reach our conclusions, while our results of the experiments and the comparison of the two implementations are given in Section 5. Section 6 consists of the Evaluation of our project. Finally, we conclude our discussion and provide some Future Work ideas in Section 7.

Chapter 2

Literature Review

Since the web is constantly exploited by attacks varying from code injection with cross-site scripting, to browser hijacking, it cannot be trusted. Spyware infection and session riding are other popular attacks able to exploit vulnerabilities especially on the browser. Many efforts have been made on preventing these attacks as well as self-propagating worms and content-based, phishing and spoofing attacks. These efforts are all trying to solve the same problem from different perspectives, leading to different contributions. Language-based approaches include static analysis, program instrumentation, and the formalization of JavaScript. In addition, auditing mechanisms for monitoring the behavior of the code have been developed. Interpreters and Web Workers have also been the focus of many studies for JavaScript isolation. Finally, browser-based approaches include Browser Operating System implementations and iframe isolation.

2.1 Language-based approaches

We can see that JavaScript isolation is a very difficult problem, since so many attempts have been made on solving it with no success. JavaScript is a dynamic language; this means that its code can change dynamically at runtime, thus violations will not be detected until the code is executed. It also contains methods like `eval`, `this` and others, that can help an attacker easily exploit it and attack the system in many ways. The first and most obvious way of preventing such attacks is static analysis; compilation of the code and error detection before it is executed. However, this ap-

proach proved to be insufficient, since methods change dynamically, or even new ones can be added. Examples of great importance using this technique are Facebook FBJS and Yahoo! ADsafe, systems widely known and used. These systems, although they were considered to be safe, were not; [MT09] discovered vulnerabilities that will be explained in 2.1.2 more thoroughly, rendering static analysis an insufficient method. Program instrumentation became the solution for many people; isolation techniques were used like filtering, rewriting, and wrapping, and error checks were inserted dynamically. However, a completely different language-based approach was also studied, the formalization of JavaScript; JavaScript type systems were designed in order to create a completely safe subset of the language.

2.1.1 Static Checks

GateKeeper

GateKeeper [GL09] is a static approach to security policy enforcement in order to protect the host page from widgets that contain malicious code. The hosting site that wants to enforce a security policy programs the policy and applies it to the newly submitted widget for restricting widget capabilities, preventing code injections and browser redirects, making sure that built-in objects are not modified, allowing cross-site scripting detection, and preventing global namespace pollution. It also uses filters for disallowing JavaScript methods like `eval`, `Function`, `with`, `setTimeout` and `setInterval`, as well as wrapping mechanisms for running runtime checks. Two subsets of JavaScript are used; $\text{JavaScript}_{\text{SAFE}}$ and the $\text{JavaScript}_{\text{GK}}$. According to which subset the code belongs to, it either proceeds to further instrumentation, or pointer analysis; a static code analysis technique that establishes which pointers point to which variables and helps the reasoning process. If the program passes the checks above and lacks statically unresolved array accesses and innerHTML assignments, it belongs to $\text{JavaScript}_{\text{SAFE}}$, otherwise, to $\text{JavaScript}_{\text{GK}}$. An important feature of this system is that it provides detailed information about why a widget fails, so that the widget developer can change the code and resubmit it. GateKeeper was tested on 8,379 real-world widgets with nine privacy policies, resulting

in only 1,341 policy violations, but unfortunately, its implementation is not available for inspection.

FBJS

Because of its great extent and influence that makes Facebook a more visible target, it needed a special JavaScript language to safely run applications in separate namespaces. Facebook applications are written in FBML, Facebook HTML. Since they need to interact with the user's profile, they are not isolated in iframes. However, their actions should be restricted. Facebook uses an isolation mechanism where every part of the application's code is rewritten in a subset of JavaScript, the FBJS [MT09], and is runtime checked to make sure it contains valid FBJS. FBJS has the same syntax as JavaScript, but in contrast with GateKeeper and ADSafe, all variable names are prefixed with a unique identifier. This way, the application cannot access native JavaScript objects. Again, unlike GateKeeper and ADSafe, the FBJS does not try to do local static analysis of field names, and access to the DOM is not allowed.

FBJS focuses on the fact that `window` should not be accessed by third-party scripts because it has global scope and access to it might cause great damage. One way of accessing `window` is by executing the identifier `this`. Since renaming `this` would change the meaning of JavaScript, `this` is replaced by `ref(this)` by the FBJS preprocessor. `ref` checks what `this` refers to, and if it refers to `window` it is rewritten to `null`. Another way of accessing the window is by accessing certain standard predefined object properties like `__parent__` and `constructor`. Therefore, these properties are blacklisted, and access to them is rewritten as `__unknown__`. Finally, properties like `valueOf` are redefined and cannot use the construct `with`.

ADsafe

Yahoo! ADsafe [Cro08] is an effort of isolating web pages from advertisements. Much like GateKeeper, Yahoo! defines a safe subset of JavaScript, filtering specific methods in order to allow guest code safely interact with the application, using a static approach. It is verified by the tool JSLint

without requiring human inspection. The script is not allowed to access the DOM directly; instead, it is given indirect access to it, by accessing an ADSafe object provided by the page's server. ADSafe does not apply script modification of any type, and can be performed at any stage. However, this approach does not prevent advertisements from accessing new methods added to the built-in prototype objects by the hosting page.

Jacaranda, Dojo Secure

Jacaranda and Dojo Secure [FWB10], just like ADsafe, are used for safe web page advertising. They have the same approach as ADsafe; they use static analyzers to verify the correctness of the code based on a specific subset of JavaScript that differs according to the system. The external script has only access to objects defined by the host, thus it cannot interact with the rest of the page. The restrictions it imposes are on the publisher, using blacklists, so that the advertisement will not breach containment. However, they both share the same vulnerability as ADsafe, where new methods added at runtime can be accessed by advertisements.

Blancura

[FWB10] shows that existing static analyzers used in ADsafe [Cro08], Dojo Secure, and Jacaranda [FWB10] are not capable of efficiently sandboxing malicious advertisements. So, it proposes Blancura, a system that whitelists known-safe properties. That way, users cannot access exploitable methods because they are not on the whitelist. The host and each guest are run in separate namespaces so that they can interact with the same objects but without interfering with one another. Like FBJS [MT09], Blancura requires that all property names begin with a unique prefix, and only these prefixes can make access requests. This way, if a vulnerable method is added by the host, the malicious script will not be able to access it. Whitelisting properties only incurs a tiny memory overhead, whereas the addition of prefixes does not affect runtime performance.

Type refinement

[KSW+13] proposes type refinement of JavaScript by implementing static analysis and taking advantage of the implicit conditional executions in order to provide improvement in analysis precision. The first condition, `isUndefNull`, checks whether a value is either `null` or `undefined` when a property is accessed, added, updated, or deleted, and returns a type error. The second condition, `isPrim`, checks whether a value is primitive rather than an object, when a value is converted into another type during execution. Finally, the third condition is `isFunc` and checks whether a value is callable. If it is not, a type error execution is thrown. [KSW+13] shows that type refinement can have a significant impact on precision of up to 86%, without causing any adverse performance impact.

2.1.2 Program Instrumentation

Improving ADSafe and FBJS

[MT09] discovered vulnerabilities in Yahoo!ADsafe [Cro08] and Facebook FBJS [MT09]. Analyzing the FBJS, they discovered that two methods could be used to return their `this` due to library leaks, thus obtaining access to the `window` object; `setSendSuccessHandler` of `LiveMessage.prototype`, and `htmlEncode` of `String.prototype`. They also discovered that runtime monitoring functions like `ref` and `idx` could be switched off, thus altering the scope of the program. These vulnerabilities could allow an attacker to gain control over the whole Facebook page, alter the user's profile and exploit browser vulnerabilities. Facebook team was informed and fixed the problem within 24 hours.

ADsafe, Yahoo! JavaScript subset, is used to safely run advertisements in a web page. While trying to prove that, [MT09] discovered the vulnerability that the JavaScript library `prototype.js` provides ADsafe code access to the global scope. This vulnerability could lead to the violation of isolation properties. Restrictions were imposed by the vendor, however, further investigation is needed since the language they proposed has the same limitations as other blacklisted static verifiers.

Combining Filtering, Rewriting, and Wrapping

[MMT09] proposes the combination of filters, rewriting and wrappers for isolating JavaScript. That way filters will prevent malicious code from executing without affecting the performance, rewriting will insert runtime checks for greater programming expressiveness, and wrappers will prevent the misuse and limit the impact of untrusted code without requiring any changes on the code. However, the rewriting process might affect performance, and wrapping mechanisms might cause runtime overhead. Assuming that a host page includes untrusted code from different programs, either benign or not, [MMT09] has two goals; the first one is restricting access to native properties making use of a `Whitelist` that consists of native object's properties that can be accessed by untrusted code, and the `Blacklist` that consists of the properties that access to them should be forbidden. The second goal is to isolate the namespace of untrusted principals, thus separate the set of global variables accessed by any two untrusted programs. The filters proposed are for disallowing all terms which contain an identifier from the `Blacklist`, the identifiers `eval`, `Function` or `constructor`, and identifiers whose name begin with $. These filters combined with rewriting and wrapper functions are designed in order to create a more expressive and safe subset of JavaScript, the $\mathrm{JS}_{\mathrm{e2}}$.

Caja

The Google Caja [Teac] is a project aiming to provide a safe subset of JavaScript for object-capability security. It is enforced by a static verifier (filter) and runtime checks (rewriting). A large JavaScript subset, Caja, compiles untrusted JavaScript code with Google Caja, and produces code filtering the `with` and `eval` methods in Cajita. Cajita is a capability-based safe subset of Caja without the `this` method, used for code evaluation. Caja allows untrusted code from different sources to interact in a safe way, however lacking in complexity and efficiency. Its alterations from JavaScript include rejection of all names ending with __ (double underscore), because access to `__proto__` of an object grants the authority to create more objects like that. Moreover, it adds the ability to freeze an object, so that its properties cannot be set, added or deleted, throwing an exception whenever such an attempt is made. When compiled, the code is separated into

modules isolated and without being able to access each other. Caja also enforces the convention that property names ending in _ (single underscore) are protected instance variables.

WebSandbox

The Web sandbox [IM09] secures web content through isolation. It builds on top of Microsoft's BrowserShield project which uses the rewriting isolation mechanism. It uses an open-sourced framework so that users can test the Sandbox by using a cross-browser JavaScript virtualization layer, and provide their feedback in the effort of providing a standardized secure web platform.

BrowserShield

BrowserShield [RDW+07]is a framework for dynamic instrumentation, built on top of Shield's vision to a new domain. The web page and any embedded script are rewritten at an enterprise firewall at runtime to use a JavaScript library that will translate them according to its policies to safe equivalents. Using the firewall, BrowserShield updates can be centralized at the firewall without having to install them. However, end-to-end encrypted traffic is not visible to a firewall, leaving the browser extension and the web publisher to handle it separately. BrowserShield is focused on HTML, script, and ActiveX controls; it is not well designed for preventing HTTP or images vulnerabilities. However, if deployed with an HTTP filter and an Antivirus, it can offer great protection, with only moderate overhead. BrowserShield can also serve as a platform for link translation, script sandboxing, and script debugging.

AjaxScope

AjaxScope [KL10], just like BrowserShield [RDW+07], is a framework for dynamic instrumentation. The web application is monitored across users, and an AjaxScope proxy acts as a mediator; it parses the code on-the-fly and rewrites it according to security policies, before it reaches the user. AjaxScope takes advantage of the instant redeployability of the web application

environment to dynamically provide differently instrumented code to the users. In order to reduce the overhead of every single user and spread it across them, the tests are distributed, and the instrumentation taking place is adaptive. AjaxScope does not require server-side modifications nor extensions or plug-ins on the browser. AjaxScope's monitoring techniques, by exploiting the power of software-as-a-service, can be applicable to a broader domain of software.

Lightweight self-protecting JavaScript

[PSC09] suggests a method of making JavaScript self-protecting and reducing its overhead. Unlike other methods of dynamic instrumentation like AjaxScope [KL10] and BrowserShield [RDW+07], the external script does not go through runtime parsing, and no code is dynamically generated. Instead, [PSC09] inserts security policies via a reference monitor in order to intercept API calls and load new code in the header of the page. Thus, its all functionality is based on security policy enforcement. Challenges to that are completeness and tamper-proofing; all API calls must be intercepted, and the code must not subvert the monitor mechanism itself. [PSC09], just like BrowserShield, does not require browser support to intercept and transform JavaScript operations. In addition, the protection can be applied either at the server-side, or at the client-side since it requires no browser modification. What is very interesting about that mechanism, is that the security policy is applied even if the code is compromised by an XSS attack.

Mutation Event Transforms

Mutation Event Transforms [ELX07], METs, is a mechanism that uses the rewriting technique to offer client-side security by enforcing security policies. These policies range from disallowing scripts in certain parts of the page, to taint-based policies that regulate the flow of credit-card information input by the user. They are specified by Web application servers as JavaScript functions and included at the top of the page. Every time a Web page is instantiated or updated, they are invoked by the client to ensure that the page conforms to the security policy before any other script is run. This way, METs transform the mutations before they take place on the web page.

METs is an easy to use system since it requires straightforward changes to existing web browsers.

CoreScript

CoreScript is an operational semantics of a subset of JavaScript, used to prove the correctness of the rewriting process studied in [YCIS07]. It was developed to be used in program instrumentation of JavaScript, combined with policy management. This combined mechanism aims at the protection against malicious client-side code, as well as patching security holes. The advantage of this tool comparing to others, is that it enables a unified framework that enforces various security policies with the same rewriting mechanism. Moreover, CoreScript is used considering higher-order scripts, scripts generated by other scripts. The script goes through security checks at runtime, guided by a customizable security policy. User prompts are generated and the web page viewer decides on how to proceed according to the prompts. Thus, there are two distinct operations described; policy management, expressed by edit automata, and the rewriting mechanism. These operations are separated using a policy interface. Although usually policies are intended to secure the hosting page from a specific attack, [YCIS07] proposes the combination of policies for guiding the rewriting process, with the purpose of battling multiple attacks at a time. Experiments on the effectiveness of CoreScript have proven successful, but unlike METs, such proxy-based mechanisms must parse data and code in requests which might lead to problems. Further investigation is needed on the practical aspects of deployment.

XPCNativeWrapper

The XPCNativeWrapper [Cen] is a JavaScript object that implements the wrapping isolation mechanism. Its aim is to provide safe access to the properties and methods of a possibly unsafe object, creating a security wrapper around it. This way, access to the properties of this object is limited. It can be used in all Firefox versions. The XPCNativeWrapper is an easy to use mechanism since it does not require much modification to the existing code. There are four case scenarios; when a protected script accesses a trusted

object, then no wrapper is created and the script has full access over the object's properties. When a protected script accesses an untrusted object, an *implicit deep* XPCNativeWrapper is created. Finally, when an unprotected script accesses either a trusted or an untrusted object, then again no wrapper is created. Firefox 3.6.2 added the `unwrap()` method for unwrapping a wrapped object. Although two bugs that were discovered in previous versions of Firefox are now fixed, it still has a lot of limitations that do not allow commonly used properties to be used with XPCNativeWrapper.

2.1.3 JavaScript type-systems

$\mathbf{JS_0}^{\mathbf{T}}$

JS_0 is a formalism of JavaScript developed by [AGD05]; it is an object-based language with features of JavaScript like dynamic addition of methods and functions creating objects. Based on that formalism, [AGD05] aim to design a type system, ${JS_0}^{T}$, to allow type inference and offer safety, so that programmers do not have to write explicitly types. [AGD05] adopts congruence for subtyping between function types, and defines a type inference algorithm, unlike [Thi05]. This is sound with respect to the type system. The challenge JS_0 is facing is the imperative nature of the language combined with the possibility of extending objects. Although it is a promising approach, further work is required in order to make the subtyping for functions more flexible.

A type-based program analyzer

[Thi05] is the first attempt at defining a type system for analyzing a weakly typed language, JavaScript. [Thi05] tracks automatic conversions that occur in JavaScript through a matching relation, and flags the suspicious ones. It includes singleton types, subtyping, and first class record labels. That way, programs can be easily rejected if they have type mismatches or suspicious conversions. Unlike [AGD05], it does not support recursive types, and no type inference algorithm is given, but there is an implementation.

Soft typing

Soft typing [CF91] combines the advantages of static typing with the flexibility of dynamic typing. Thus, it is a dynamically typed language that detects potential type errors statically. It focuses on a core functional language similar to ML and Scheme, the prototypical representatives of static and dynamic languages; it is an extension of the ML language that supports union types, recursive types, and parametric polymorphism. It also uses a type inference engine with an algorithm that inserts dynamic runtime checks, but does not reject programs. Further work on adding type intersection to the polyregular types in order to subsume them would be beneficial.

Object-capability model

[MMT10] studies the object-capability model where the programming language objects are both subjects that initiate access and objects of regulated actions. This model is another approach for restricting interactions between web applications, without preventing them from interacting with the user or the hosting page. [MMT10] is interested in *authority safety*, a subset of the object-capability goals that provides safety conditions that support isolation without any enforcement techniques. [MMT10] proves that capability safety implies authority safety, and focuses on the Cajita subset.

2.2 Auditing mechanisms

Apart from Language-based techniques, monitoring systems have been developed for auditing JavaScript behavior and alerting users in order to prevent possible attacks.

2.2.1 Intrusion Detection System

The first attempt of designing an auditing mechanism for monitoring JavaScript code and logging operations is discussed in [HV05]. This approach uses an Intrusion Detection System, IDS, for detecting malicious code. The detection can take place either on anomaly detection, comparing

the behavior of the script to the 'normal' behavior of other scripts, interpreting deviations from the 'normal' behavior as the problem, or on misuse detection, comparing the operation of a script to some pre-defined attacks called signatures. It audits JavaScript code and detects possible attacks. Its overhead increases as does the number of operations logged, so future work is required on decreasing the overhead by using more efficient I/O buffering techniques as well as using more sophisticated signatures.

2.2.2 SABRE

Since extensions are not constrained by the same origin policy, they are executed with the privileges of the browser, thus they can misuse these privileges. Even when they are benign, they still share the browser's vulnerabilities and therefore can be exploited by a malicious web site. SABRE [DG09] is a Security Architecture for BRowser Extensions that monitors JavaScript extension's behavior; it differs from prior work in that it does not reject malicious code, but instead it uses information flow tracking to analyze plug-ins. Its core functionality is based on identifying extensions that carry sensitive information, and it manages that by using labels. In case the object that carries sensitive information is accessed in an unsafe way, an alert is raised. Another difference from prior work is that the monitoring process happens within the browser and not at the system level.

Its implementation though, requires some changes in the web browser. SABRE can successfully identify information flow violations, and despite the high overhead it reports, the performance does not slow down. It has proved to be a substantial mechanism, but future work is required for making it browser-portable, and for determining whether it can leverage static analysis techniques in order to reduce its high overhead.

2.3 Interpreters

Interpreters are JavaScript engines that parse, compile, and execute JavaScript code. Examples of commodity browsers implementations involve SpiderMonkey, deployed by Mozilla, V8, deployed by Google and Opera, JavaScriptCore, deployed by Safari, and others. These implementations can

be used combined with various techniques to offer secure execution of third-party scripts in web pages.

2.3.1 Safe interpreters

The goal of a safe interpreter is triple; access control in various objects, ensure independence of different contexts, and manage trust relationships. By doing that, it offers both data security to the user, and user privacy. [AM98] suggests that security issues should be considered during the initial design of new scripting languages. Safe interpreters tackle cross-window and Trojan horse attacks. Since objects with browser or window data are read accessible, safe interpreters isolate scripts from executing unsafe commands, making use of a framework in which a variety of security policies can be implemented. They alert the user to suspicious behavior and terminate trust relationships when unloading of HTML documents.

2.3.2 js.js

[TBK12] proposes js.js, a JavaScript interpreter that runs on top of JavaScript and allows third-party scripts to be executed within a sandboxed environment. Thus, code is double protected by both the interpreter and a wrapper around it. js.js is browser-portable, and protects against page redirection, spin loops, and memory exhaustion by placing optional checks inside the interpreter loop. The important feature of js.js is that it supports the full JavaScript language unlike static analyzers previously discussed.

Its core functionality is based on allowing the application to have fine-grained control over what actions a third-party script can perform. The application acts as a mediator that intercepts all access requests from the script, deciding whether it will allow them or not before they reach the browser, while using the js.js library to execute the third-party script in the sandbox. However, js.js presents much overhead and future work is required.

2.4 Web workers

Web Workers are JavaScript scripts that run in separate threads, executed from an HTML page. Although their initial aim was to enable web pages

to be responsive even when they run long tasks, they can also be exploited and used in an attempt to make web pages safe.

2.4.1 Treehouse

TreeHouse's [IW12] aim is to isolate third-party scripts and limit their influence by using Web Workers as containers to run guest code. It is a system that tries to minimize modification or redesign, development-time and runtime code changes, and server configuration. It requires no browser modifications, but one of its limitations is that the guest code sometimes needs minor restructuring. It is the first work on virtualizing the browser in a backward compatible way that requires no server configuration and protects against exhaustion attacks. TreeHouse gives the application author fine-grained control over the code by letting him define what access is permissive by guests.

TreeHouse virtualizes the browser's API to the sandboxed code. Each application runs in a different Web Worker. These Workers do not have access to the DOM; instead, a broker is installed in each Web Worker. In the application environment a monitor is required. The role of the broker is the virtualization of the browser's resources (virtual DOM), and the handling of communication between the script and the monitor, using message passing. The monitor is used for applying changes in the VDOM to the real DOM and for delivering DOM events that are decided by the author to guests. When the guest code invokes the browser's API, modifies the DOM, or makes a request for registration to DOM changes, the broker decides on whether these actions will be allowed and forwarded to the DOM according to specific policies. If they are not, the broker terminates the guest.

TreeHouse's implementation was successfully run on Chrome, Safari, IE10 and Firefox. However, it demonstrates significant overhead on DOM operations, with much latency shown in page load on large applications. Finally, porting an application to TreeHouse requires modest effort by changing some lines of code, whereas writing a non-trivial policy requires about 30 minutes.

2.4.2 JSandbox

JSandbox [Gre] is an open source JavaScript sandboxing library that makes use of HTML5 web workers in order to run untrusted content safely. It has been successfully tested on Firefox and Google Chrome.

2.4.3 Bawks

Bawks[The] is a JavaScript sandbox based on JSandbox from [Gre]. It hosts untrusted code inside a Web Worker thread, and uses cross-origin messaging for communication. It makes use of four different functions; `Load` that loads the script, `Call` that calls the function, `Eval` that evals the code inside the untrusted scope, and `Whitelist`, a list with all the trusted functions.

2.5 Browser-based approaches

2.5.1 Browser Operating System

Tahoma

Tahoma [CHGL06] is a browser-portable system that uses virtual machines for isolating web applications, so that users do not need to trust the web browser. It is based on the Browser Operating System, BOS, a software layer on which browsers execute. Not only does it provide strong isolation between web applications, but between the browser and the host system as well. Tahoma is a proposal of complete re-examination of the browser architecture and is easy to use since it requires only three modifications on the browser.

When a browser instance wants to run an application, the BOS acts as the mediator, checking whether it conforms to the network policy; if it does, it is executed on the virtual screen and finally aggregated to the physical. Each browser instance can only execute a single web application, unlike in conventional browsers.

It can effectively prevent both sandbox and spoofing vulnerabilities since it uses virtual machines for sandboxed environments, and the window manager decorates the browser instance which cannot be later modified. How-

ever, it presents deficiencies in sharing interfaces and improper labeling vulnerabilities; although it limits the sharing interfaces they still exist, and although web services declare the scope of their web applications, Tahoma depends on external systems like DNS, and if these systems are subverted, then Tahoma will be subverted as well. However, it is a system that offers strong isolation without sacrificing performance.

IBOS

The IBOS [TMK10], Illinois Browser Operating System, is both an operating system and a browser. Like Tahoma, it uses the BOS architecture, but its principle goal is to reduce the Trusting Computing Base (TCB) for the browser. It manages to remove almost all traditional OS components and services by mapping browser abstractions to hardware abstractions. This mapping happens by pushing security decisions to the lowest layers in order to avoid millions of lines of library and OS code. IBOS is the first attempt of its kind to improve browser and OS security. Its principle is to control the sharing interfaces among web applications and traditional applications, like Tahoma, as well as enforce security policies without changing the web applications. It also avoids OS sandboxing because it is complex and difficult to implement.

Its performance in security safety is effective. As for page latency, although it reduces tremendously the TCB, it does not have the results expected. When comparing it to Tahoma, which operates mostly on hardware-level abstractions, we find that the latter is unable to provide full backwards compatible web semantics from the VMM and more fine-grained protection for browsers.

Google Chrome

Google Chrome [RBP09] improves its security by modifying its architecture. Its major components are two; the high-privilege browser kernel, and the low-privilege rendering machine. The former is trusted and acts with the user's authority to provide network access, store cookies and history databases, and draw the user's interface. On the other hand, the rendering engine is not trusted to interact with the user's operating system and other

processes since it parses HTML, executes JavaScript and performs other tasks for the Web pages. Google Chrome's architecture consists of many layers in order to prevent this interaction; first, the untrusted web content is sandboxed within a JavaScript virtual machine which protects different sites from each other. The next layer consists of OS and runtime exploit barriers to make it more difficult to exploit vulnerabilities in the JavaScript sandbox. Finally, a sandbox is used again at the operating-system level in case exploits escape the previous layers. This Windows implementation runs with a restricted Windows security token, an invisible Windows desktop, and a restricted Windows job object. Compared to TreeHouse which manages to isolate scripts within web applications, Google Chrome isolates web applications only from each other. It also starts with a complex system and tries to remove unneeded portions of it, while IBOS starts with a clean slate and builds on top of it.

Internet Explorer

Internet Explorer [Zei], just like Google Chrome, is a commodity browser with its own architecture for security. More specifically, IE8 has incorporated a new feature called Loosely-Coupled IE or LCIE which is a collection of internal architecture changes in order to improve the reliability, performance, and scalability of the browser. The browser frame and its tabs are isolated and located in separate processes and components use asynchronous communication with each other, so that potential failures in a tab cannot affect the rest of the browsing session. Moreover, the frame and the broker object are located in the same process which improves performance. Finally, Low and Medium integrity tabs can reside in the same UI frame; this way, Protected Mode can be turned on or off on a per-tab basis, thus vastly improving usability.

The OP browser

Another web browser, the OP browser [GTK08], was designed in order to improve browser security. The OP browser combines operating system design principles with formal methods, and is partitioned into smaller subsystems. These are the web page subsystem, a network component, a user-interface

component, a storage component, and a browser kernel. All communication between the subsystems is managed by a browser kernel that enforces security features. OS-level sandboxing techniques are used to limit the interactions of each subsystem with the operating system by denying file system and network access. Security policies are enforced to be applied to browser plug-ins by interposing on message passing in the browser kernel. The OP browser provides better security and fault isolation than monolithic browsers, and among with the Gazelle browser [WGM+09] they are the only ones to offer the same protection to plug-in content as to standard web content. However, the fact that it uses standard browser modules to provide the DOM tree renders it inefficient [MD11].

The Atlantis browser

[MD11] propose the Atlantis browser, a microkernel web browser, not only for better security but for a more extensible execution environment as well. The Atlantis kernel defines a narrow API for basic services like network I/O and screen rendering. Its goal is to enforce the Same Origin Policy and to allow web developers to customize the runtime environment for their pages.

At the bottom of the Atlantis architecture is the master kernel which consists of the switchboard process that creates the isolation containers for web pages, the device server that allows access to peripheral devices, and the storage manager that provides a key interface.

The Atlantis browser differs from Gazelle [WGM+09], the OP [GTK08], and IBOS [TMK10] in that it has a single master kernel and multiple sandboxed per-instance kernels so that even if the kernel is compromised, the entire browser will not be compromised, unlike the others.

The Gazelle browser

Gazelle is a secure web browser introduced in [WGM+09]. The browser kernel runs in a separate protection domain and interacts directly with the underlying operating system. The rest of the principals communicate with one another only through the browser kernel. It uses the Same Origin Policy to offer consistency across various resources. Gazelle has the same principal instance as Google Chrome's [RBP09], but with one difference; Google

Chrome considers the sites that share the same registrar domain name to be from the same site, whereas Gazelle considers them different.

The Gazelle browser offers strong isolation between web domains, but does not protect intra-domain components like Atlantis does [MD11]. Some of the problems it is facing are display protection and resource allocation.

Multi-process Browser architecture

[RG09] shows that web content can be divided into separate web programs without losing compatibility with existing content, and that separate instances can exist within the browser. The purpose of [RG09] is to create a multi-process browser architecture for web application isolation in order to improve the browser's robustness and performance. OS processes are used as an isolation mechanism, and web program processes are sandboxed in order to help the enforcement of some aspects of the browser's trust model. [RG09] design idea is that one process is dedicated to each program instance and the components that support it, while the remaining components are safely executed in a separate process. The architecture described has been implemented by Google in the Chromium [RBP09] web browser.

2.5.2 Frame isolation

Fragment Identifier Messaging and postMessage

In [BJM09] security policies for frame isolation like the *same origin policy*, the *permissive policy*, the *window policy*, the *descendant policy*, and the *child policy* are discussed and compared, answering the question of how to select the best navigation policy that would defeat both security and compatibility issues. Trying to solve the problem of frame isolation in mashups, where interframe communication is required, they end up proposing the *descendant policy*, adding two techniques. Since navigation is essential for interframe communication, collaborating with the HTML 5 working group, their proposed techniques are the `fragment`identifier messaging and the `postMessage`. The former is a channel that uses frame navigation to send messages between frames directly to each other, without using the network's latency. This way, the attacker cannot eavesdrop the message. The

`postMessage` technique is a browser API that allows asynchronous communication between frames. These techniques were deployed by commodity browsers like Internet Explorer, Google Chrome, Safari and Firefox.

SMash

SMash [DKBS+08] mitigates gadget hijacking by monitoring the frame hierarchy for unexpected navigations. Although these navigations cannot be prevented, the user will be alerted. There are however cases where the attacker can lure the victim into entering sensitive information, since SMash waits 20 seconds for a gadget to load before assuming that it has been hijacked.

Subspace

Subspace [JW07] uses a multilevel hierarchy of frames so that the `document.domain` property communicates directly in JavaScript. Subspace also uses the descendant navigation policy to prevent gadget hijacking.

OMOS

Open Mashup OS (OMOS) [ZYG08] is a framework for secure communication in Mashup applications. Its design goal is to be compatible with all mainstream browsers and easy to use and understand by mashup developers. It guarantees mutual authentication, data confidentiality, and message integrity for Mashlets; client side components that run in the browser.

Its security communication protocol relies on the Same Origin Policy (SOP) so that DOM elements, events, and cookies are protected, the URL property of an iframe is write-only, and partial change of URL is not allowed.

AdJail

AdJail [TLGV10] is a framework for addressing security threats posed by third party advertisements. Publishers are able to specify confidentiality and integrity policies on advertisements, and targeting scripts that select which ads to be displayed have restricted access to sensitive data. AdJail

is also compatible with ad network billing operations like click metrics, and guarantees consistency in user experience. It is easy to adopt, and works on major browsers without any modifications.

Ads are fetched and executed in a hidden sandbox and all communication takes place as if no interposition happens. At first the policy denies any access to the script unless the publisher grants the ad any kind of permission.

AdSentry

Untrusted ads are sandboxed using a JavaScript engine mediating their access to the page with AdSentry [DTLJ11]; a flexible isolation framework. Access control policies can be specified by both web publishers and users for regulating the behavior of the advertisements.

The difference in AdSentry's approach is that a shadow JavaScript engine is used for executing untrusted ads and making sure that they will not affect the rest of the page. It exposes the full spectrum of JavaScript functionality and it is strictly sandboxed. No modification to the browser is required, and the protection is achieved only with a small performance overhead.

2.6 Others

MashupOS

MashupOS [WFHJ07] is a browser-based multi-principle operating system that focuses on abstractions for communication and protection of web browsers. Its goal is to allow communication without the compromise of confidentiality and integrity. MashupOS introduces two new HTML tags for integrators to include unauthorized content.

- `<Sandbox>` : for private unauthorized content that belongs to the integrator
- `<OpenSandbox>` : hosted by any domain

When the script is enclosed in the `sandbox` tag and comes from a different domain it cannot be accessed (read or write global objects, invoke

script functions, modify DOM elements) by the enclosing page; it can only be accessed when it comes from the same domain. In contrast, for the `OpenSandbox` tag, no matter which domain hosts the script, its content can be fully accessed by the page.

Although the code inside the sandbox cannot follow references outside of the sandbox, data references from within the sandbox can be used by the outside of it.

OMash

OMash [CHC08] is an access control model for writing secure mashup applications that allows objects to communicate only via their declared public interfaces. It is inspired by MashupOS but it does not use the Same Origin Policy because SOP relies on insecure services and has design limitations [CHC08]. Moreover, instead of expressing different trust relationships with different abstractions, OMash proposes a single abstraction for expressing all trust relationships. It only requires a few preferences to be set.

BEEP

Browser-Enforced Embedded Policies (BEEP) [FHEW08] allows web developers to define a whitelist of scripts that may run in a page. However, this approach is targeted at isolating static content and does not apply to interactive mashup applications.

SpoofGuard

Web spoofing attacks manage to lure victims into revealing sensitive information by masquerading to a trusted web site. SpoofGuard [BML+07] is an Internet Explorer browser plug-in developed to prevent such attacks; it monitors web pages for spoof attacks, and when such attacks are discovered the user is warned. It manages to do that by computing a spoof index; if that index exceeds a level selected by the user, the web page is considered malicious. SpoofGuard also uses history to check whether the user has visited the page before. Web spoofing is a special case of intrusion detection, but it additionally has access to both honest and spoof pages, which means

it has a better chance of catching the attack. However, it has the downfall of producing false alarms and that its tests can be fooled, so it needs further improvement.

ConScript

The hosting page can use ConScript [ML10], the first general browser-based policy enforcement mechanism for JavaScript, to express security policies that are enforced at runtime. These policies can be automatically generated either by server-side code through static analysis, or by client-side code through runtime analysis. Although modifications on the browser are easy and small, they are required.

BFlow

BFlow is proposed in [YNKM09] for observing confidential data that flows into, out of, and within the browser, preventing it from leaking. It is a browser security system that uses a reference monitor in the browser to enforce information flow control. One of its design goals is to only allow the human user and the origin web site to see the information derived from the data when confidential data arrives from a web site, unless the site specifically allows it to go to another web site. Its second goal is to mark confidential data as confidential, unless the site allows the removal of confidential marking.

Mugshot

Developers use Mugshot [MEH10], a system that captures every event in an executing JavaScript program, in order to collect traces from programs and gain visibility into the application execution. That way, developers can replay past executions and improve performance evaluation.

In case content from an application does not pass through the proxy that Mugshot uses to reproduce the load events, faithful replay is not guaranteed. Moreover, unexpected interactions between HTML, CSS, and JavaScript, or wrong installation of browser plug-ins by the user may cause a bug. However, Mugshot introduces little overhead at logging time.

SIF

Servlet Information Flow (SIF) [CVM+07], is a software framework for building web applications. It uses new language features to enforce confidentiality and integrity policies. SIF controls the flow of confidential and low-integrity information to clients, and it also enables users to protect information from one another. [CVM+07] shows that application-defined mechanisms for access control and authentication can be integrated securely with language-based information flow and makes an important step towards wider use of information-flow control.

Chapter 3

JavaScript Overview

JavaScript [Fla02] is a web scripting language with object-oriented capabilities. It enables programs to interact with the user and dynamically create HTML content so that web pages are not static anymore. JavaScript resembles C, C++, and Java syntactically, however, type specification is not required. It is an interpreted language inspired by Perl. It was originally called LiveScript [Fla02], but its name was changed to JavaScript at the last minute.

The most common variant of JavaScript is client-side JavaScript; when a JavaScript interpreter is embedded in a web browser. This refers to the scripting ability of the interpreter of the language, combined with the Document Object Model (DOM) [Fla02].

3.1 Previous uses

JavaScript is a general purpose language:

What JavaScript *can* do [Fla02]

- it can be embedded within any application
- it can be used for writing programs to perform arbitrary computations
- it can control the document appearance and content

- it can control the browser behavior
  - pop up dialog boxes
  - open new browser windows
  - frame layout
  - move forward and back
  - download and display content
- interact with HTML forms
- interact with the user
  - define event handlers
- read and write cookie values
- control browser content
- allow content to be dynamically generated
- produce image rollover and animation effects
- interact with Java applets
- build delays or repetitive actions
- provide information about the name and version of the browser
- provide information about the color and size of the monitor

What JavaScript *cannot* do [Fla02]

- no graphics capablities
- no reading or writing of files
- no networking support
- delete data or plant viruses
- the `value` property of HTML `FileUpload` cannot be set

3.2 Features-exploits-dynamism

As we have already mentioned, JavaScript is a dynamic language. This dynamism renders it an easily exploited language that attracts many attackers.

One thing that JavaScript's features allow, is to execute annoying pop ups. However, users have the ability to restrict this pop up abuse by advertisers.

Moreover, if the value property of the HTML `FileUpload` could be set, any kind of file could be uploaded to the server.

Same Origin Policy (SOP): JavaScript relies on the SOP. When iframes are included, interactions between the different iframes is limited by this security restriction. Only the properties of the documents that have the same origin (protocol, host, and port of the URL) as the document that contains the script can be read.

Cross-site scripting (XSS): server-side web developers defend against cross-site scripting attacks; when HTML tags or scripts are injected into a website. These kind of attacks take place when document content is generated dynamically.

Example:

Greet the user by name

```
<script>
var name = decodeURIComponent(window.location.search.sub
string(1)) || "";

document.write("Hello " + name);
</script>
```

Such a page is invoked with a URL like:

`http://www.example.com/greet.html?Efi`.

and the text displayed is `Hello Efi`.

However, an attacker could inject a malicious script into another, invoking the URL:

```
http://siteA/greet.html?name=\%3Cscript/
src=siteB/evil.js\%3E\%3C/script\%3E.
```

In this case, site B includes a link to site A that injects a script from site B. The script `evil.js` is now embedded in site A and it can manipulate A's content. It can also read cookies stored by site A.

In order to prevent such attacks, the HTML tags should be removed. In our case:

```
name=name.replace(/</g,"\&lt;").replace(/>/g,"gt;");
```

replacing all angle brackets with their corresponding HTML entities.

Denial of Service: when a user visits a website with JavaScript enabled, his browser might be at risk since the SOP does not protect against brute force - denial of service attacks. Infinite loops of computations or alert boxes can slow down the CPU or render the browser unresponsive. Methods like `setInterval()` can attack the system by allocating lots of memory.

Chapter 4

Overview of the various solutions

4.1 Pre-requisites

This type of work, testing the code after applying different changes on it, requires an available web page for printing out the results. So, our first task was to create a website under the University of Bath directory, change the permissions and make it look for the files in the correct directory. This took us approximately 40 minutes.

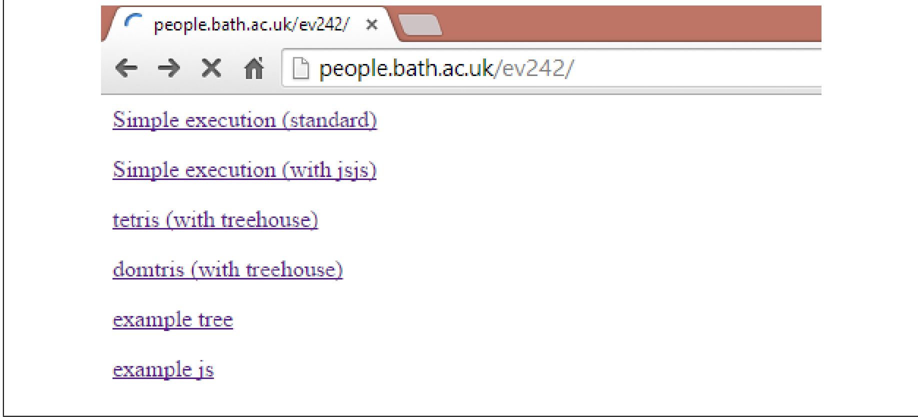

Figure 4.1: Our home page

Since the `public_html` directory for our web page and the essential files were stored under our University's account, we used FileZilla for file manipulation (read, write, over write, delete etc.) while connecting to the

University's servers. Installing this software takes about 15 minutes.

In order to investigate both the TreeHouse and the js.js sandboxing implementations we had to undertake a JavaScript tutorial from w3schools. After approximately 10 hours of studying and coding in JavaScript and HTML, we were ready to start experimenting in an advanced level.

4.2 TreeHouse

4.2.1 Initial Experiments

Hands on the right piece of code

After reading the paper from Lon Ingram and Michael Walfish about TreeHouse, we were pointed to the actual repository on github [Ing]. This included different demos in HTML format, JavaScript scripts, some required packages, the base and default policies, and other template policies. Saving the code locally, including all the files, required only a couple of minutes.

We started an HTML page from scratch, under our web page, that would run under TreeHouse with the appropriate `tag`, trying to output some simple JavaScript code in order to ensure that we were looking at the right piece of code. The whole process took about one hour and the results were not satisfying. After trying unsuccessfully to print out various things and reading more thoroughly both the paper and the rest of the files (30 minutes), we found out about domtris; domtris is a JavaScript Tetris clone that uses the DOM to render the game and handle user input.

What is allowed and what is not

Without having any expertise in JavaScript, we experimented for an hour trying to figure out what is allowed and what is not by printing out a script inside the head and inside the body. That way we could check if it makes any difference where the script is inserted. We also tried to redirect the page and we found out that the page gets redirected whether the appropriate command is inside the head or inside the body tag.

When experimenting with alert events and page redirections for 3 hours, although at first we concluded that domtris does not catch alerts, we finally

realized that in order for the TreeHouse sandbox to be enabled, we need to include the treehouse tag when sandboxing the script. So, we came to the conclusion that a script without the treehouse tag allows page redirections and alerts, but a script with the specified tag does not.

```
<script src="../demos/tetris.js"
  type="text/x-treehouse-javascript"
  data-treehouse-sandbox-name="worker1"
  data-treehouse-sandbox-children="#tetris">
</script>
```

Example of `tetris.js` with TreeHouse functionality

After two hours of reading about policies, we realized that the scripts in domtris do not have any policy included, so we had some questions to answer. Is TreeHouse coming with a policy by default? If so, which one? TreeHouse comes with two policies; the base policy and the default policy, but should we include them in order to work or do they come with TreeHouse anyway?

Since alerts and page redirections are only captured by using the treehouse tag and without using any policies, we were determined that at least one of them must do so. More to that, when we tried to create a new policy, we found out that the treehouse tag would not let us see any visible results, and we were concerned that a possible problem would be the fact that the `tetris.js` application did not have any functionality, since when we would load the domtris page we were not able to actually play the game.

Can the user set what to be allowed?

When trying to experiment with methods that are and are not allowed by policies we did the following. It took us about half an hour to create a policy that would not allow another user to change the background color. We managed to change the background color to red without the policy successfully. On the contrary, when we included a policy (with the appropriate tag that comes with it), the background color stayed the same. What was obvious

to us, was that our up until then effort with regards to policies was a total success.

However, while reading the paper once again for about 15 minutes, we decided to create a new HTML page from scratch and experiment on this one in order to double check our results. This page included two buttons; one that when the user clicks on it, a function is triggered that is wrapped inside a script with the treehouse functionality, and one without treehouse. Having in mind that in order for an action to take place both the base and the default policy must allow it, we made the following attempts:

At first we changed the policy to allow `onclick` events (1 min), and dom manipulations like `innerHTML` and change of color (2 mins). We realized that when the script is wrapped with treehouse nothing happens.

Our second attempt was to change the policy to allow `postMessage` events (1min), but the results were the same. The first thing that crossed our minds was that something was up with the monitor, and that led us to our third attempt.

Our last attempt before coming to a meaningful conclusion was to change the policy in order to allow background change of color, but this time in the domtris page, so that the monitor is implemented. This took us about 3 minutes, but again nothing happened.

The important conclusion that we were led to after many different attempts of manipulating the dom, was that changes to the page only work when they are inside a script that does not include any treehouse functionality. Since the treehouse tag is supposed to be translated into something meaningful by the monitor, that made us think that our real problem was with the monitor. Looking through the code once again, we realized that the necessary packages for the monitor to work were missing from the directory. We had to cease operations on TreeHouse and contact the author in order to provide us with the appropriate packages. This is not a rare phenomenon according to [CC13] where authors are contacted and asked for source code very often. This whole process is best described by the term **reproducibility**: "*only if my colleagues can reproduce my work should they trust its veracity.*" [CC13]

4.2.2 After contacting the author

We were lucky enough to get feedback from the author who provided us with a new and updated repository and instructed us to do the following:

- Update our clone
- Check out the working domtris-demo branch
- Run a webserver on port 8080 from the root of our clone
- Load `http://localhost:8080/demos/domtris.html`.
- Click in the gameboard of the DOMTRIS page and then press space to start

We followed these instructions step by step (1 hour) until we finally were able to run the application locally on our apache server with full functionality of the game and start experimenting on the domtris page.

What is allowed and what is not

Our first attempt after the successful loading of the game was to write a script in the domtris page that would execute an alert, and after loading the page we indeed got the alert. When we wrapped that script with treehouse, we neither got an alert nor the game was working. We had the same results when we tried to redirect the page using the location API. At this point we were not confident whether the monitor was finally indeed functioning correctly.

One hour later, we tried the same things, but this time inside the script that imports the `tetris.js` file. The results were still the same with the difference that the game was running.

After another hour of efforts, we inserted the same code that would redirect the page inside the `tetris.js` file. In order to check whether it is important where the redirection happens in the code, we inserted it both in the `startGame()` and in the `gameOver()` functions. The page did not get redirected and the game stopped loading the moment it read the appropriate command (either at the beginning or at the end). Even when we wrote a new policy that would allow everything, we still were

not successful. When we inserted the code `startGame()` at the end of the `gameOver()` function, the game started again after it was over, which gave us evidence that changes in the `tetris.js` file under treehouse are possible. However, we still were not able to change the background color (`document.body.style.backgroundColor="\#00FF00"`;).

We came back to the 'example' page that we had created, the one without the tetris application, and we used the location API in order to reload the page with the code `location.reload()`;. This action only worked when the script was not wrapped with treehouse.

Can the user set what to be allowed?

At this point, we started re-thinking about the policy issue. We knew that if the author does not include a policy on his own, his application under treehouse would ship with the base policy that cannot be overwritten, and the default one. So, we imported them in the page using their url location, but the results were still the same.

A couple of days later, the author informed us that the issue was that a proxy for the location API was not yet implemented in the demo. Workers have read-only access to the location object, but to provide functionality like `location.reload,`code must be written to send a message to the parent page asking that it perform the reload, and that he had not written that code yet.

Now the question was:

Is the location API ***only*** *facing this problem?*

We had to experiment with other methods as well in order to answer that question.

When we inserted the code `game.innerHTML=navigator.appCodeName`; the text `Mozilla` appeared in the game field (**Figure 4.1**). We also tried with `game.innerHTML=location.host`; and this time `localhost:8080` appeared in the game field. These were our first clues that manipulation of the DOM under treehouse was possible. In order to be more certain about DOM manipulation, we experimented with the background color once again. We found out that `game.style.backgroundColor="\#00FF00"`; would successfully change the color of the game field.

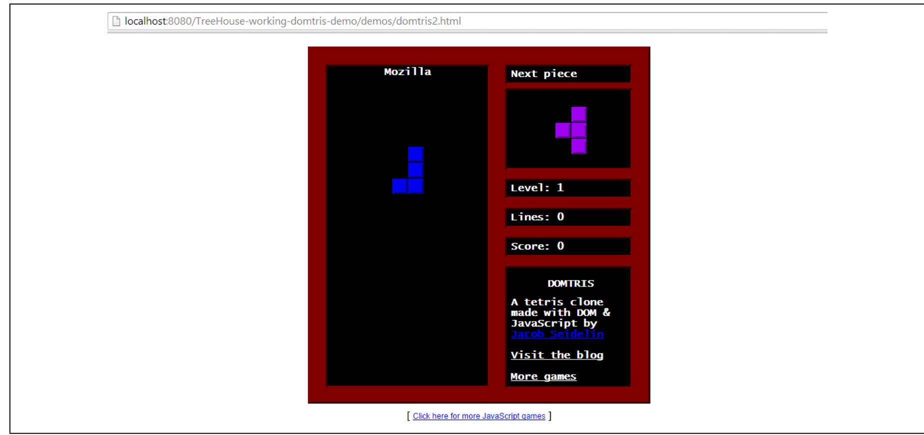

Figure 4.2: TreeHouse: DOM manipulation (innerHTML)

What `navigator`, `location`, and `backgroundColor` had in common was that all three were events allowed to be changed by the user under treehouse's default policy.

How can the user set what to be allowed?

Since at this stage we were successful, we decided that the next step would be to try to manipulate an element not allowed by the default policy, in order to check how well the policy functions. We concluded on the `onclick` and `onmouseover` events. The code `game.onclick=function(){game.style.backgroundColor="\#00FF00"};` did not affect the background color when run under treehouse. On the contrary, the color was changed when the script was not running under treehouse. Fortunately, we had the same satisfying results with `game.onmouseover=function(){game.style.backgroundColor="\#00FF00"};`. This led us to the conclusion that the default policy was working 100% right.

The only thing left to do was to change the policy our application was running under. After about two hours, we created one that would allow everything, another one were nothing was permitted, and a few others with various alterations on the dom. In every case the results were the same; it was as if the default policy was still implemented on the domtris page, and the other policies were not functioning for some reason. The conclusion we

reached was that the `setPolicy` system that was used to create the policy was not working. That did not allow any other policy to be implemented, so the application was always running under the default one.

4.3 JavaScript in JavaScript

In order to include the API in our website, we had to include either the minified library (the translated shared compiled library) or the non-minified one with the wrapper script. We chose the latter since it was useful for debugging and modifications. This process took us about 5 minutes.

Hands on the right piece of code

We found many examples on github to experiment with. We chose the 'simple execution' one which was the simplest to understand and work on. This example implements the addition of two numbers inside a wrapped function.

Our first experimentation was to change the inputs of the addition function in order to check whether the application works, and it did. This led us to our next step; changing the input types in order to check the correctness of the output. The variable that was bound to the sandboxed environment before any of our changes included the function: `src="nativeAdd(1,1)"`.The changes we made were the following:

- When a character was added to a number, the output returned was that same number.
  - **Input:** `src="nativeAdd('a',1)`";
  - **Output:** 1
- When a character without the string tag was added to a double number, the output returned was an error because *a* is a variable name that cannot be added to any number, thus the value could not be converted to a number.
  - **Input:** `src="nativeAdd(a,1)"`;
  - **Output:** Can't get result because an error happened.

- When instead of a function the variable included a simple number, the output was that same number.
  - **Input:** `src="5"`;
  - **Output:** 5
- When the first argument was a number, and the second a minus one, the output was the addition of those two numbers.
  - **Input:** `src="nativeAdd(5,-1)"`;
  - **Output:** 4
- Having as arguments two characters, the output returned was 0.
  - **Input:** `src="nativeAdd('a','b')"`;
  - **Output:** 0

These experiments showed that the expected output was returned according to the input.

What is allowed and what is not

After about 2 hours of experimentation, we tried to redirect the page to facebook in order to check whether page redirection is allowed, with the code:

```
window.location.replace("http://www.facebook.com");
```

inside the wrapped function. When that code was added inside the function and after the `return` statement, we had no visible results. However, when the code was placed before the `return`, the page got redirected. We figured out that this was expected since when the function reaches the `return` method, it terminates.

After reading the [TBK12] paper, we found out that there is a helper function that specifies the expected types of the arguments of the wrapped function and what the wrapped function should return. Thus, this is why the page redirection could happen. But the question was: Should it happen?

Another concern of ours was whether we should only worry about what is inside the `src` or what is inside the wrapped function?

The meaningful conclusions that we managed to reach after about 2 hours of thorough investigation of the code and reading of the paper were that the example code in the `wrapper.js` file was just an initialization tool that we had to configure.

What the application does and what is allowed

At first we realized that the `JSJS.Init()` method creates a new runtime with 8 mb of memory, sets the javascript version, an error reporter, creates an empty global object in the interpreter space, and initializes the standard javascript global objects(classes) like `Array`, `Date`, and `String`. In order to ensure that every action is allowed since the global object is empty, we did the following:

- Experimentation with alert (`window.alert(str)`):
  - In a different script: we got the alert
  - Inside the script before the sandbox initialization: we got the alert
  - Inside the script after the sandbox initialization: we got the alert
  - Inside `nativeAdd()`: we got the alert
- Experimentation with prompt (`window.prompt(str,str)`)
  - In a different script: we got the prompt
  - Inside the script before the sandbox initialization: we got the prompt
  - Inside the script after the sandbox initialization: we got the prompt
  - Inside `nativeAdd()`: we got the prompt
- Experimentation with redirection (`window.location.replace(url)`)
  - In a different script: we got the redirection
  - Inside the script before the sandbox initialization: we got the redirection

  - Inside the script after the sandbox initialization: we got the redirection
  - Inside `nativeAdd()`: we got the redirection

- Experimentation with background color (`document.style.backgroundColor='color'`)
  - In a different script (before and after the sandboxed script): no change of color
  - Inside the script before the sandbox initialization: the color changed
  - Inside the script after the sandbox initialization: the color changed
  - Inside `nativeAdd()`: the color changed

- Experimentation with innerHTML (`document.getElementById(id).innerHTML="str"`)
  - In a different script after the sandbox: nothing
  - In a different script before the sandbox: nothing
  - In a different script inside a called function: we got the change
  - Inside the script before the sandbox initialization: we got the change
  - Inside the script after the sandbox initialization: we got the change
  - Inside `nativeAdd()`: we get the change

As it was expected, all the methods we experimented with were allowed by the wrapper file. Cases where we did not have the expected results like experimentation with the background color of the page and `innerHTML` were because the code was invoked from a function of a different script, and that function had to be called somehow.

The wrapper file initializes the `document` element and the `window` element, creating the `window` object, the `window.alert`, and the `window.prompt` methods. We were not sure whether these methods should

be 'locked' in order to be impossible for a user to use them, but the experimentation so far showed us that actions on them are fully allowed.

However, since `page redirection`, `alert` and `prompt` use the `window` element, and the `background color` uses the `document` element, we had to experiment with another element in order to be sure that the empty global object allows everything, and not just the elements that have been initialized in the wrapper file.

- Experimentation with reload (`location.reload()`)
  - In a different script: we got the reload
  - Inside the script before the sandbox initialization: we got the reload
  - Inside the script after the sandbox initialization: we got the reload
  - Inside `nativeAdd()`: we got the reload

The results were the same; everything was allowed. Our next thought was to try to lock the window method and trigger a window prompt in order to check whether the element gets locked and cannot be accessed.

- Experimentation with `SetLock`.
  - `JSJS.SetLock(window,'locked')`: we got the prompt
  - `jsObj=JSJS.SetLock(window,'locked')`: we did not get the prompt, but we did not get any output
  - `jsObjs=JSJS.SetLock(document,'locked')`: we did not get the prompt, but we did not get any output
  - `jsObjs=JSJS.SetLock(document,'unlocked')`: we did not get the prompt, but we did not get any output

After the above experimentations we realized that the application stops whenever it reads the `SetLock` method, so up to this point we were not able to lock any methods.

Since we knew that the `wrapper.js` file was a tool that had to be configured, we decided to manipulate its content. At first, we changed the file

to an empty one in order to make sure that it affects our output and it did, since there was no visibility of results after running the addition function. After that, we removed the function that creates the `window.prompt` event in order to check whether `prompts` are still allowed, but we could still execute a window prompt. We did the same with the `window.alert` method and we had the same results. However, that was in fact expected since other methods that were not initialized in the wrapper file were allowed. Moreover, when we changed the expected arguments of the `window.alert` method to either `null` or to `objPtr` instead of `charPtr` just to check whether any error messages would be triggered, we could still get the alert without any error messages.

The methods that the js.js implementation has initialized and are executed in the sandboxed space when invoked by the user are:

- `document.getElementById()`;
- `window.top.location`;
- `window.alert()`;
- `window.prompt()`;

To conclude, the `wrapper.js` file is indeed a file where methods and functions are initialized and are invoked by the host page scripts so that they are safely executed inside a sandbox. However, we were not able to set specific methods *allowed* or *not* since the functionality where the user has the option to setup white/black lists of browser elements is not yet implemented and is left as future work [TBK12].

Chapter 5

Experimental results

5.1 Evaluation

Just like the authors did, we evaluate TreeHouse on its latency overhead and its ease of use, and JS in JS on its latency overhead. Our experiments run on a Lenovo ThinkPad Edge with a 2.60 GHz Intel Core i5 processor and 4 GB of RAM running Chrome 37.0.2062.120, Firefox 32.0, and IE 11.0.9600.17278 under Windows 8 Operating System.

5.1.1 TreeHouse

Page load latency: We evaluate TreeHouse on the latency of the DOMTRIS page load with and without the TreeHouse functionality. In order to do that we include a script in the head of the page to get the starting time with the code:

```
<script type="text/javascript">
    start=(new Date()).getTime();
</script>
```

Then we create the `latency` function in order to calculate the loading time, and we put the result in a `<p id="loadingtime"></p>`.

```
function latency() {
   var end = (new Date()).getTime();
   var sec = (end-start)/1000;
   var p = document.getElementById("loadingtime");
   p.innerHTML = sec;
}
```

Finally, we load the `latency` function in an onload event that is triggered when the page finishes loading.

```
window.onload = function () {
    pageload();
}
```

DOMTRIS with TreeHouse

An example of the latency page load evaluation on Chrome is depicted in the 5.1 image.

Figure 5.1: Page load latency on Chrome

In more detail, we perform 10 runs in each browser, and the results in milliseconds are shown in **Table 5.1**:

Google Chrome	Mozilla Firefox	Internet Explorer
69.0	184.0	88.0
24.0	253.0	87.0
24.0	13.0	97.0
24.0	121.0	35.0
27.0	100.0	49.0
73.0	153.0	38.0
26.0	161.0	34.0
21.0	124.0	40.0
22.0	135.0	54.0
26.0	161.0	38.0

Table 5.1: DOMTRIS page load latency with TreeHouse in milliseconds

DOMTRIS without TreeHouse

In order to run DOMTRIS without TreeHouse, we had to remove every treehouse functionality where the type of the script was:

```
type="text/x-treehouse-javascript"
data-treehouse-sandbox-name="worker1"
data-treehouse-sandbox-children="#tetris"
```

Including the monitor:

```
<script src="../lib/require.js"></script>

<script>
  // web worker compat
  window.doc = document;
  window.win = window;

  require({
     baseUrl: '../src',
     packages: [{
         name: 'jsdom',
         location: '../lib/jsdom/lib',
         main: 'jsdom'
     },
     {
         name: 'node-htmlparser',
         location: '../lib/jsdom/node-htmlparser'
     },
     {
         name: 'underscore',
         location: '../lib/underscore',
         main: 'underscore'
     },
     {
         name: 'treehouse',
         location: './'
     }]
    }, [ 'kernel' ], function (kernel) {
       console.debug('Kernel loaded.');
       kernel.initialize();
       console.debug('Kernel initialized.');
     });
</script>
```

Again, the results of 10 runs in milliseconds are shown in the table below:

Google Chrome	**Mozilla Firefox**	**Internet Explorer**
17.0	40.0	61.0
16.0	84.0	12.0
16.0	34.0	45.0
14.0	34.0	36.0
17.0	36.0	71.0
13.0	63.0	37.0
14.0	62.0	12.0
14.0	42.0	10.0
16.0	39.0	40.0
16.0	13.0	10.0

Table 5.2: DOMTRIS page load latency without TreeHouse in milliseconds

Table 5.2 reports the results of the mean values of with and without TreeHouse for each browser in milliseconds. Google Chrome shows the best results in both cases. Mozilla Firefox seems to be the slowest of the three both with and without TreeHouse. In every case, the time of the page load without TreeHouse is almost half the time of the one with the TreeHouse functionality.

Experiment	**Chrome**	**Firefox**	**IE**
DOMTRIS, baseline	15.3	44.7	33.4
DOMTRIS, TreeHouse	33.6	152.2	56.0

Table 5.3: Mean values in milliseconds

Ease of use: Developers' job is to integrate TreeHouse into a web application and write policies [IW12]. The integration part consists of including the monitor JavaScript code into the web page and the appropriate `treehouse` tag in the script type. Writing policies can either be easy or difficult depending on the complexity of the policy, the understanding of how TreeHouse operates, and how familiar one is with coding in JavaScript.

After thorough experimentation and reading of the [IW12] paper, and after taking a JavaScript tutorial, we managed to write a couple of policies, spending only a couple of minutes.

```
setPolicy({
   '!api': {
        '*' : true
    },
   '!elements': {
        '!attributes': {
            '*' : true
             },
        '!tags': {
            '*': true
        }
    }
});
```

Example 1: Everything is allowed

```
setPolicy({
    '!elements': {
        '!attributes': {
             '*': {
                 onmouseover: true,
                 onclick: true
              }
         }
 }});
```

Example 2: `onmouseover` and `onclick` events set to true

5.1.2 JS in JS

Page load latency: We evaluate the time required for the start up and shut down of the js.js implementation, as well as the overhead of a simple execution. More specifically, we evaluate the overhead of the `NewRuntime`, `NewContext`, `GlobalClassInit`, and `StandardClassesInit` (they all belong in the start up), `DestroyContext`, and `DestroyRuntime` (they both belong in the shut down), and finally the `EvaluateScript` function. In order to do that, we had to include some extra lines of code in the `wrapper.js` and the `simple-js.html` files for the start up - shut down and the simple execution accordingly.

NewRuntime

```
var NewRun = Date.now();
var rt = JSJS.NewRuntime(8 * 1024 * 1024);
var NewRun2 = Date.now() - NewRun;
reportMessage("NewRuntime time: " + NewRun2);
```

NewContext

```
var NewCon = Date.now();
var cx = JSJS.NewContext(rt, 8192);
var NewCon2 = Date.now() - NewCon;
reportMessage("NewContext time: " + NewCon2);
```

GlobalClassInit

```
var Glob = Date.now();
var global_getter = JSJS['PropertyStub'];
...
var Glob2 = Date.now() - Glob;
reportMessage("GlobalClassInit time: " + Glob2);
```

StandardClassesInit

```
var InStan = Date.now();
var init_standard = JSJS.InitStandardClasses(cx, global);
console.log("init standard classes " + init_standard);
var InStan2 = Date.now() - InStan;
reportMessage("InitStandardClasses time: " + InStan2);
```

DestroyContext

```
var DestCon = Date.now();
JSJS.DestroyContext(jsObjs['cx']);
var DestCon2 = Date.now() - DestCon;
reportMessage("DestroyContext time: " + DestCon2);
```

DestroyRuntime

```
var DestRun = Date.now();
JSJS.DestroyRuntime(jsObjs['rt']);
var DestRun2 = Date.now() - DestRun;
reportMessage("DestroyRuntime time: " + DestRun2);
```

EvaluateScript

```
var evalScr = Date.now();
var rval = JSJS.EvaluateScript(jsObjs.cx, jsObjs.glob, src);
var evalScr2 = Date.now() - evalScr;
reportMessage("Simple execution 1+1 time: " + evalScr2);
```

The results of 10 runs for each browser in milliseconds were the following:

NewRuntime results:

Google Chrome	Mozilla Firefox	Internet Explorer
14.0	19.0	41.0
17.0	14.0	20.0
14.0	14.0	14.0
16.0	13.0	23.0
16.0	19.0	20.0
7.0	16.0	22.0
15.0	14.0	23.0
7.0	17.0	17.0
16.0	14.0	20.0
7.0	24.0	18.0

Table 5.4: NewRuntime time load in milliseconds

NewContext results:

Google Chrome	Mozilla Firefox	Internet Explorer
30.0	37.0	241.0
36.0	31.0	140.0
29.0	32.0	141.0
30.0	31.0	133.0
28.0	31.0	138.0
29.0	30.0	124.0
28.0	27.0	134.0
32.0	37.0	127.0
30.0	33.0	131.0
25.0	46.0	124.0

Table 5.5: NewContext time load in milliseconds

GlobalClassInit results:

Google Chrome	Mozilla Firefox	Internet Explorer
8.0	23.0	7.0
8.0	18.0	7.0
8.0	20.0	7.0
8.0	21.0	6.0
9.0	21.0	9.0
2.0	22.0	7.0
9.0	22.0	7.0
4.0	22.0	6.0
9.0	24.0	7.0
3.0	29.0	11.0

Table 5.6: GlobalClassInit time load in milliseconds

StandardClassesInit results:

Google Chrome	Mozilla Firefox	Internet Explorer
89.0	120.0	84.0
86.0	123.0	101.0
98.0	112.0	106.0
95.0	105.0	90.0
86.0	105.0	101.0
44.0	115.0	91.0
89.0	107.0	100.0
86.0	170.0	94.0
90.0	118.0	98.0
81.0	156.0	94.0

Table 5.7: StandardClassesInit time load in milliseconds

Execute 1+1 results:

Google Chrome	Mozilla Firefox	Internet Explorer
75.0	92.0	93.0
76.0	86.0	106.0
87.0	98.0	106.0
74.0	102.0	121.0
77.0	94.0	107.0
11.0	97.0	90.0
80.0	106.0	118.0
6.0	107.0	105.0
79.0	101.0	102.0
7.0	137.0	98.0

Table 5.8: Execute 1+1 time load in milliseconds

DestroyContext results:

Google Chrome	Mozilla Firefox	Internet Explorer
33.0	47.0	39.0
35.0	44.0	57.0
36.0	52.0	44.0
32.0	44.0	37.0
36.0	47.0	39.0
18.0	42.0	38.0
34.0	52.0	66.0
21.0	43.0	52.0
36.0	57.0	52.0
19.0	58.0	54.0

Table 5.9: DestroyContext time load in milliseconds

DestroyRuntime results:

Google Chrome	Mozilla Firefox	Internet Explorer
6.0	3.0	7.0
3.0	4.0	10.0
6.0	5.0	8.0
4.0	8.0	10.0
4.0	5.0	8.0
3.0	7.0	6.0
5.0	6.0	6.0
2.0	4.0	9.0
4.0	5.0	8.0
4.0	6.0	8.0

Table 5.10: DestroyRuntime time load in milliseconds

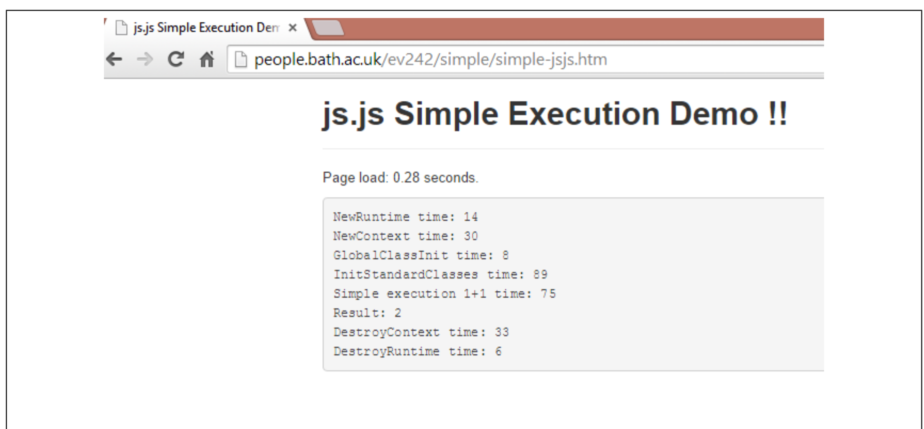

Figure 5.2: Page load latency on Chrome in milliseconds

Table 5.11 shows the mean time in milliseconds across 10 executions required to execute the start up and shut down routines for the js.js implementation, as well as the simple execution `1+1`. Google Chrome presents the least overhead in every execution. Internet Explorer appears to have the biggest overhead when loading the page.

Experiment	**Chrome**	**Firefox**	**IE**
NewRuntime	12.9	16.4	21.8
NewContext	29.7	33.5	143.3
GlobalClassInit	6.8	22.2	7.4
InitStandardClasses	84.4	102.5	95.9
SimpleExecution	57.2	102.0	104.6
DestroyContext	30.0	48.6	47.8
DestroyRuntime	4.1	5.3	8.0
Page load	247.8	378.8	475.2

Table 5.11: Mean time load in milliseconds

We created the **Table 5.12** by making some calculations based on the **Table 5.11**. We added the mean times of `NewRuntime`, `NewContext`, `GlobalClassInit`, and `InitStandard`Classes in order to calculate the whole start up time. We then added the mean times of the `DestroyContext`, and `DestroyRuntime` in order to calculate the whole shut down time. What we deduced from our results was that the quickest routine is the shut down in every browser. Although creating the runtime environment is the slowest procedure, it still is not an expensive cost.

Experiment	**Chrome**	**Firefox**	**IE**
Start up	133.8	174.6	268.4
Simple execution	57.2	102.0	104.6
Shut down	34.1	53.9	55.8

Table 5.12: Mean time load of start up, simple execution, and shut down routines in milliseconds

5.2 Comparisons

This section is about comparing the two investigated implementations both to each other (Section 5.2.1) and to the authors' results (Section 5.2.2). They will be compared for their purpose, methodology, and contributions.

5.2.1 Head-to-head comparisons

Purpose

Although TreeHouse and js.js use a different methodology, they both have the same purpose; allow site operators to control included code by providing a sandbox where JavaScript code can be run by a web application. TreeHouse's additional goal is to be a mechanism that works today by minimizing browser modification, redesign, development-time code changes, run-time code changes, and server configurations, and to allow the access and the resources for the contained script to do its job. On the contrary, js.js is designed to be generic, easy to use, and flexible, and is used to bind any kind of global object inside the sandbox space.

Methodology-design

Both TreeHouse and JS in JS rely on *isolation*, *interposition*, and *virtualization*. As shown in the **Table 5.13**, TreeHouse uses Web Workers for isolation, whereas js.js uses JavaScript Interpreters. For interposition and virtualization, TreeHouse uses a broker in each worker that virtualizes the browser resources, and a monitor that runs in the JavaScript environment of the window in which the user loaded the application. On the contrary, js.js relies on a Mediator for interposition that uses a js.js library to execute a third-party script in a sandbox.

In TreeHouse, what is permitted is decided by the application author using policies, whereas js.js does not include any policies. Communication between the monitor and the guest script in TreeHouse takes place with message passing.

Both implementations provide full JavaScript support and require no browser changes. Fine-grained control is only supported in js.js. Finally,

TreeHouse's limitations include that its Trusted Computed Base includes the browser and is not small, the guest code sometimes needs minor restructuring, and that the future of Web Workers is uncertain. On the other hand, js.js makes it very complex to create a virtualized DOM since there is no functionality for a user to setup white/black lists of browser elements, sites etc.

	TreeHouse	**JS in JS**
Isolation	Web Workers	JavaScript Interpreters
Interposition	Broker Monitor	Mediator
Virtualization	Broker	Mediator
DOM Permission	Author Policies	Author wrapper file
Communication	Message passing	N/A
Limitations	browser in the TCB minor code restructuring Web Workers future	complex VDOM no white/black lists
Full JS Support	✓	✓
Fine-grained control	✗	✓
No Browser changes	✓	✓

Table 5.13: TreeHouse and js.js design differences

Contributions

TreeHouse:

- "*Applying the operating systems ideas of sandboxing, virtualizing, and resource management to JavaScript.*" [Ing]
- "*The design of TreeHouse, which instantiates these OS ideas without browser modification.*" [Ing]
- "*The implementation and evaluation of TreeHouse.*" [Ing]

JavaScript in JavaScript:

- "***Fine-grained control:*** *Rather than course-grained control, e.g., disallowing all DOM access, an application should have fine-grained control over what actions a third-party script can perform.*" [TBK12]
- "***Full JavaScript support:*** *The full JavaScript language should be supported, including* ***with*** *and* ***eval****, which are impossible to support with static analysis.*" [TBK12]
- "***Browser Compatibility:*** *All major browsers should be supported without plugins or modifications.*" [TBK12]
- "***Resilient to attacks:*** *Resilient to possible attacks such as page redirection, spin loops, and memory exhaustion.*" [TBK12]

5.2.2 Comparisons to prior work

In this section we compare the authors' results to ours.

TreeHouse

Table 5.14: Author's

	Chrome	**Firefox**	**IE**
without	24.0	12.0	6.0
with	361.0	181.0	405.0

Table 5.15: Ours

	Chrome	**Firefox**	**IE**
without	15.3	44.7	33.4
with	33.6	152.2	56.0

Table 5.16: TreeHouse: Comparison of results

We can see that our results differ from the author's. Without TreeHouse their mean times are not so different from ours, since we are talking about milliseconds, but our best browser seems to be Google Chrome, whereas for them Google Chrome's results are the worst. When TreeHouse functionality is enabled, Mozilla Firefox appears to be their best option, presenting almost half the overhead of the other two browsers. On the other hand, we found out that Firefox is the slowest of the three, presenting mean times of 3 to 4

times of the other two. These results can be justified since prior experiments were made on very old versions of the browsers and a lot might have changed.

JS in JS

Here we compare the author's results to ours; however, it is not known which browser was used for their experiments. The `GlobalClassInit` routine on the IE shows a very large overhead compared to both the other browsers and the author's results. Firefox shows a large overhead in `StandardClassesInit` and `Execute 1+1` compared to their results as well. Generally, their results are very close to ours.

Table 5.17: Author's

	N/A
NewRuntime	25.2
NewContext	35.8
GlobalClassInit	15.5
StandardClassesInit	60.1
Execute 1+1	70.6
DestroyContext	33.3
DestroyRuntime	1.8

Table 5.18: Ours

Chrome	**Firefox**	**IE**
12.9	16.4	21.8
29.7	33.5	143.3
6.8	22.2	7.4
84.4	102.5	95.9
57.2	102.0	104.6
30.0	48.6	47.8
4.1	5.3	8.0

Table 5.19: JS in JS: Comparison of results

Chapter 6

Evaluation

Level of difficulty: The completion of this project required about four months. In this section we describe how difficult this project was to our perception, the hours we spent, and our opinion on the results.

- In order to successfully complete this project, at first we had to make a thorough investigation of previous work and different solutions. We found 49 papers that each one proposed a different approach and we included them all. Finding papers based on prior work was an easy task since the JavaScript isolation problem is widely known and worries most developers and users.
- After that, we created a website where we could apply our experiments, which only required some basic level knowledge of HTML.
- The next step was to find the code of each of the two implementations we would evaluate. As instructed on both papers, the code was easily found on the appropriate repositories on github, so we did not have to contact the authors.
- Following that, we stored it into the right directories and loaded it into our website.
- The first major difficulty we faced was inexperience with the JavaScript language, so we had to undertake a tutorial from w3schools and read a book about JavaScript [Fla02] which slowed down the whole procedure.

- Starting with `js.js`, we soon realized that the code was working and what we had to do was configure the `wrapper.js` file, so we left it aside and continued with the TreeHouse code.

- TreeHouse was not as easy as we found `js.js`. It took us a while to realize the reason why we had no functionality of the Tetris game (under the DOMTRIS page), so we had to contact the author in order to provide us with the right piece of code. As we said before, although this is something usual when it comes to reproducibility, we did not see that coming and we had to cease operations on this tool for quite a while.

- But this was not our only problem with TreeHouse; neither the full code for the demo was yet implemented, nor the policy system was working. By the time the author informed us about these problems, we had already spent more hours experimenting on TreeHouse than it was planned.

- While waiting for the author's replies we got back to `js.js`. Altogether, the only major problem we faced with `js.js` was with comparability; we evaluated its overhead on Google Chrome, Mozilla Firefox, and Internet Explorer. However, we could not compare our results to the author's, since we did not know which browser they used for the exact results.

- The overhead evaluation of both implementations was not a difficult task since it only required some JavaScript code in order to print out timing results for each method we wanted to measure.

- Finally, it is worth mentioning that every step we took including hours spent was written down as a diary, making the whole process a lot easier.

Hours spent: Table 6.1 shows in more detail the hours we spent on each part of the project. Previous work consists of 49 papers we used as literature review, spending on average two hours on each paper. We needed 40 minutes to create our webpage under the University of Bath servers, and 5 minutes to find and download the code of each implementation. Our meetings were 4, accounting for 3 hours and 55 minutes in total. The time spent on studying JavaScript programming, its methods, and ways to be exploited was 12 hours and 30 minutes. Experimentation on TreeHouse required 18 hours and 30 minutes, whereas on JS in Js 11 hours and 45 minutes. More time was spent on TreeHouse because it was the first one to experiment on, and because it appeared to be more complicated due to the problems we faced. Finally, evaluating TreeHouse took us 7 hours, while only 30 minutes were spent on evaluating JS in Js. Again, we spent about 6 hours trying to figure out how to make the evaluation happen (print timing results), and since TreeHouse was the first we evaluated, it was also the one we spent more time on.

	TreeHouse	**JS in JS**	**Generic**
Previous work	-	-	98h
Webpage	-	-	40m
Code download	5m	5m	-
Supervisor meeting	-	-	3h 55m
Study JavaScript	-	-	12h 30m
Experimentation	18h 30m	11h 45m	-
Evaluation	7h	30m	-
Writing	-	-	63h 26m
Total	25h 35m	12h 20m	178h 35m

Table 6.1: Hours spent on the whole project based on hour diary

Knowledge gained: The goal of this project was to become more familiar with research and practical aspects of security. We understood how big the problem of hosted applications can be, and how much JavaScript can be exploited due to its dynamism and exploitable features. We read so many papers in order to learn all the different approaches one could take to eliminate this problem, and we decided to give our attention to sandboxes. By reproducing the results of prior work we now have the knowledge of how

sandboxing implementations can be integrated into applications, how they work, how a developer can manipulate the code so that it will not later be manipulated by a third-party, and how to evaluate such implementations according to how safe they are, how much overhead they cause, and their ease of use. More to that, we learned the JavaScript language in a more advanced level.

Progress results: As mentioned in the proposal of this project, in the beginning of May 2014, work to be done consisted of:

- Code installation and measurement
- Desk exercise
- Set of metrics
- Evaluation

with 9 weeks as best case scenario, and 15 weeks as worst case scenario. The project's lifetime was 17 weeks due to difficulties faced in the meantime. Problems with the code appeared that were not expected (with TreeHouse), so we had to cease operations twice and wait for the author's guidance for about 20 days in total. More to that, we experienced difficulties communicating with each other due to the fact that for 1 month we lived in different continents so it was almost impossible to find a time that suited us both. So, lack of communication made the progress of the project a bit slower. However, if we deduct these delays, the project was completed as it was expected having very satisfying results.

Experimental results: The reason why we chose to investigate the TreeHouse and JS in JS implementations, is because they represent two important categories of sandboxing; Web Workers and JavaScript Interpreters. Both of these methods sandbox external scripts in a new and interesting way, so for this reason results based on them would be, and proved to be very significant to our research.

Methodology: Our methodology was both appropriate and effective. The steps we followed (JavaScript tutorial, installation of the code, understanding of how the implementations work by thorough reading of the

papers, decision upon the set of metrics, testing and experimenting with the code, evaluation of the code) led us to the kind of the results we were expecting.

Chapter 7

Conclusion

7.1 Summary

A lot of research so far has been focused on making web sites and hosted web applications that are written in JavaScript secure. JavaScript's dynamism has rendered it an exploitable scripting language that needs to be isolated so that even if malicious attackers try to inject code or cause any kind of harm, it will not affect the rest of the code.

A thorough investigation of the various approaches of securing JavaScript has proved that this is a very big problem that has not been fully solved. Different approaches have managed to partially solve the problem, but even in that case, there is still room for further improvement.

The goal of this project was to evaluate two implementations that try to solve the problem from different perspectives, TreeHouse and JS in JS, by reproducing prior work, and to determine how effective they are.

The Treehouse product, as available on `<github>`, does not, as far as we could tell after correspondence with the author, correspond to all the claims made in [IW12]. When we downloaded the code there was no functionality of the demo (the tetris game under DOMTRIS), so we had to contact the author in order to point us at the right direction. As already mentioned in this project and in [CC13], there are many examples where authors are being approached because the code they claim to have uploaded is nowhere to be found, or does not function (our case). This project, although not intended to, now belongs to this category. Moreover, although our results on overhead for the different browsers for TreeHouse differ from the author's,

our experiments tell us that web developers can use such an implementation in order to safely include third-party scripts to their web page without sacrificing performance. However, if they do wish to set their own allowed methods, work needs to be done on the `setPolicy` system of the TreeHouse policy system. In other words, TreeHouse is an effective tool for JavaScript isolation that does not sacrifice performance, but our experiments, for now, show that what can and what cannot be manipulated by other applications is pre-configured.

As for JS in JS, our results on overhead were very close to prior experiments, but their's did not cope with our project's aim, comparability, since we did not know on which browser overhead was tested. Finally, just like with TreeHouse, further work needs to be done in the `wrapper.js` file so that it becomes more than an initialization tool and it allows for users to set their own white/black lists of methods.

While finishing this project we realized that the JavaScript isolation problem is indeed a very controversial problem that attracts many and different solutions contributing partial solutions to different aspects of the problem.

7.2 Future Work

We faced a lot of problems in the process of TreeHouse's evaluation. We had to run it under our own server listening to a specific port, we later realized that proxies for every `API` were not yet implemented in the demo, so the methods that we could experiment with became even less. Finally, when we were to write policies in order to check how a developer can set the methods he wants to be manipulated by other applications, we were not successful since the `setPolicy` system required for the creation of a policy was not working. Therefore, since evaluating the intention of TreeHouse appeared to be a difficult task, we believe that there is room for TreeHouse's improvement.

As for `js.js`, neither reproducibility was a smooth task since they do not mention which browser they used for the overhead results, nor the evaluation of setting which methods to be allowed, since that was left for future work.

In the future we could evaluate other implementations both from the

same categories and from different ones, so that we have a larger sample of comparisons which will lead to us reaching to more accurate conclusions.

Appendix A

Project Diary

This diary was maintained by the author during the project. One aim was to quantify the hours spent on experimentation, JavaScript etc. (but not general project tasks, paper reading etc.), in order to substantiate the analysis in the project of how time-consuming the various tasks of *using* this technology were.

15/5/2014

- Download TreeHouse master code: **5 mins**
- Download js.js master code: **5 mins**

16/5/2014

- Meeting with my supervisor - start with js.js (twitter application): **15 mins**

17/5/2014

- Reading js.js paper: **50 mins**
- Download and install node.js: **2 - 3 mins**
- Continue reading: **30 mins**
- Run the shell: **10 mins**

19/5/2014

- Meeting with my supervisor - create my website and include the code in my public_html directory: **40 mins**

20/5/2014

- Make my website readable (chmod 711 ev242): **1 min**
- Include lib.minO2.js in my directory: **5 mins**
- Make lib.minO2.js executable: **5 mins**
- Meeting with my supervisor - make the website look for the SUBdirectory, 1+1 demo: **30 mins**
- Fix the buttons: **15 mins**

21/5/2014

- Study JavaScript exploitance: **60 mins**
- Changed the output (simple execution): **5 mins**
- Page redirect (simple execution): **5 mins**
- Study: **1h 30 mins**

  40 mins:

- `src=nativeAdd(character,double)` outputs null
- `src="5"`, outputs 5 // it considers the second argument as 0
- `src=nativeAdd(-5,6)` outputs 1
- `src=nativeAdd(a,`b) outputs null
- `src=nativeAdd(window.location.replace("http://facebook.com"),5)`, outputs nothing

22/5/2014

- try different things: **1 h**

  // improved the without jsjs implementation

  //when nativeAdd returns d1+d2 everything is ok. when we add a page redirection AFTER the return, everything is ok. when we add a page redirection BEFORE the return, we get the redirection.

  //i think that after the return, the function ends, that's why we don't have the redirection AFTER the return.

  //the helper function specifies expected types of ARGS and RETURN, that's why the page redirection can happen. BUT should it happen???

  //is our only concern what is inside the 'src' aka the third party script, or what is inside the function as well (in this case nativeAdd)?

- `src=window.location.replace("http://facebook.com")`, redirect to facebook

29/5/2014

- change double to bool: return "0" `->` false: **1 min**

- meeting with my supervisor: **1 h 30 mins**

  //figured out that the example code is just an initialization tool and that I need to check the code and configure it (shouldn't the example on the paper of the author catch page redirection by default?)

6/6/2014

- read the Treehouse paper again: **30 mins**

- created a link to tetris.js running under treehouse, changed it to hello-guest, still nothing as an output. Inserted some js code inside the script, still nothing: **45 mins**

- meeting with my supervisor (found out about domtris after trying unsuccessfully to print out various things): **30 mins**

  // require is like import in Java

// doesn't catch page redirection

***1 hour**

- tried a script of printing out inside the head
- tried to run a script inside the head, inside a function and then loading the function
- tried page redirection inside the head, inside a function -> the page got redirected

  //stopped because my webpage stopped refreshing

7/6/2014

***total of 3 hours**

//domtris doesn't catch alert
//No! actually it does!! when the script is sandboxed with treehouse, we get no page redirection and no alerts!

- tried a simple javascript script, once without treehouse(alert, page redirection), and once with treehouse (no alerts or page redirection) -> success!
- js.js: couldn't print out with reportMessage but fixed it! (included the appropriate javascript scripts)

8/6/2014

- read about policies and how to change them in treehouse: **2 h**

  //the scripts I run and test do not use any policy, is treehouse coming with one by default? if so, which one? there is default policy and a base policy, but should I include them in order to work or do they come with treehouse anyway? at least one of them must do so, because possible attacks like alerts and page redirections are captured by only using the 'treehouse' tag (i mean without including any policy)

- i tried to make another policy but i cant check it because the tetris application is not working, and the treehouse tag doesn't let me do a thing: **10 mins**

--- TRAVEL TO GREECE ---.

19/6/2014

- created a policy with a background colour set to false: **30 mins**
- changed the background colour in domtris to red and it worked (it turned to red without the policy and it didn't change with it): **10 mins**

--- JavaScript TUTORIAL ---.

17/7/2014

- finished the JavaScript tutorial from w3schools and the quiz - ready to start working on the code I guess: **10 h**

19/7/2014

- read the paper once again: **15 mins**
- created a new html page with simple html - javascript: **2 mins**

  //alerts are on (without treehouse)

  //(in order for an action to take place, both the base and the reference policy must accept it)

  //base policy: everything is accepted, BUT XHRs must only open asynchronously

  //reference policy: is used by default, when no other policy is used by the author

2 hours in total

- **1st attempt:** changed the policy to allow onclick events, dom manipulations(innerHTML, change of colour)

  //work outside treehouse BUT inside the script nothing happens

- **2nd attempt:** changed the policy to allow postMessage event: nothing happened

  //!!! my guess is something's up with the monitor;

- **3rd attempt:** changed the policy to allow background change of color in the domtris (so that the monitor is implemented): nothing happened

  changed the order `->` worked! it depends on which one goes first

21/7/2014

2 hours

- inserted the code: `document.write` to see what happens - the url at the end of the page disappeared

- removed the whole style tag so we don't get confused with all the graphics - nothing happened

  //***conclusion: changes to the page only work when they are inside a script that does not include any treehouse functionality -¿ the new policy does not work, something is up with the default

- tried again with the background color, setInterval and alerts -¿ same conclusion

`---` WAITING FOR THE AUTHOR AND START WRITING THE THESIS (contents etc) `---`.

29/7/2014

3 hours

- experiment again with everything I've done so far regarding js.js -> same results
- read the paper

  // !!! stuck !!!

30/7/2014

1 hour

back to domtris!! the author replied to me!!

so, he provided me with a new and updated repository and instructed me to follow these specific steps:

1. update your clone
2. check out the working-domtris-demo branch
3. run a webserver on port 8080 from the root of your clone
4. load http://localhost:8080/demos/domtris.html.
5. click in the gameboard of the DOMTRIS page and then press space to start

I did all these and it worked! It took me about one hour to figure everything out though.

now I run the application locally on my server - did some experimentations **(1 hour)**:

1. wrote a script for an alert - I got the alert
2. wrote a script with the treehouse tag for an alert - i didnt get the alert AND we had no functionality of the game
3. wrote a script for page redirection - the page got redirected

4. wrote a script with the treehouse tag for page redirection - the page didn't get redirected AND we had no functionality of the game

// is this a success, or do we still have the same problem with the monitor??

1 hour

- I tried the same things, but this time inside the script that imports the tetris.js file: i didnt get neither a page redirection nor an alert, BUT we had functionality on the game
- tried to redirect the page inside tetris.js -¿ the game stops at the exact spot where the appropriate command is found (start, ending)
- wrote a new policy that allowed everything without any success

3/8/2014

1 hour

- inserted an alert after the game is over - nothing happened
- inserted the startGame (so that the game would start again) after it was over - it worked!
- tried with backgroundColor and bgColor - nothing
- used the tetris-pageload-policy (postMessage is only allowed) and the game stopped when it got to that command
- experiment on the 'example' page (without the tetris app)

  //location.reload (only works without treehouse)

  30 mins

  //so, scripts under treehouse CANNOT manipulate the dom (because web workers can't). they can import scripts, create child workers, and issue XHRs

5/8/2014

30 mins

- included the default.js and base-broker.js in the html page - nothing

6/8/2014

2 hours

//the author told me that there is something wrong with the proxy and the location API that needs to be fixed

- i tried with navigator but i had the same results
- wait!! I inserted the code: `game.innerHTML=navigator.language;` inside the `startGame()` function and the text 'undefined' appeared in the game field
- I inserted the code: `game.innerHTML=navigator.appCodeName;` and the text 'Mozilla' appeared in the game field!
- the same happened with `game.innerHTML=location.host;` (localhost:8080)
- the backgroundColor works too!!!!! super excited! i typed: `game.backgroundColor="#00FF00";` and nothing happened
- but then i typed: `game.style.backgroundColor="#00FF00";` and it worked!!!
- I changed the backgroundColor using the domtris1 policy (where everything is allowed) and it worked!!
- bad news. i changed the dotris policy so that nothing is allowed and the color still changed

  //that means that I am not good with writing new policies..

  //stuck again - i am gonna try something that is not allowed by the default policy

- ok!! inside `startGame()`: `game.onclick=function(){game.style.backgroundColor="#00FF00"}; ->` the color changed without the treehouse tag, it didn't change with the tag (onclick is not allowed by the default)
- i had the same satisfactory results with onmouseover
- i copied the default policy and i created a new one(domtris2) replacing every false with true `->` no!domtris2 is a fail (generally i cant create a right policy)

7/8/2014

20 mins

- changed the `backgroundColor` in the default policy to false `->` the game stopped working because this method is used by the application
- changed the `onclick` and `onmouseover` in the default policy to true `->` nothing happened (i guess the default policy cannot be over written)

  //it does not listen to any policy other than the default!! something is seriously wrong

11/8/2014

4 hours

1. the JSJS.Init() creates a new runtime with 8mb of memory, sets the javascript version, an error reporter, creates an (empty) global object in the interpreter space, and initializes the standard javascript global objects(classes) like Array, Date etc
2. experimentation with alert:
   - in a different script `->` we get the alert
   - inside the script before the sandbox initialization `->` we get the alert
   - inside the script after the sandbox initialization `->` we get the alert

- inside `nativeAdd()` `->` we get the alert

3. experimentation with prompt:
    - in a different script `->` we get the prompt
    - inside the script before the sandbox initialization `->` we get the prompt
    - inside the script after the sandbox initialization `->` we get the prompt
    - inside `nativeAdd()` `->` we get the prompt

4. experimentation with page redirection (`window.location.replace`):
    - in a different script `->` we get the redirection
    - inside the script before the sandbox initialization `->` we get the redirection
    - inside the script after the sandbox initialization `->` we get the redirection
    - inside `nativeAdd()` `->` we get the redirection

5. experimentation with backgroundColor (`document.style.backgroundColor`):
    - in a different script (before and after the sandboxed script) `->` no change
    - inside the script before the sandbox initialization `->` changed
    - inside the script after the sandbox initialization `->` changed
    - inside `nativeAdd()` `->` changed

//a window.alert method is created under customAlert(str), wrapped, and defined

//a window.prompt method is created under customPrompt(prom, defText), wrapped, and defined

// maybe we need to create a method just like alert or prompt and unlock it

// or not

//* up until now everything is allowed (alert, prompt, redirection, backgroundColor)

// alert, prompt, and redirection use the window method (i guess that's a coincidence) AND the background color uses the document element

6. experimentation with reload (`location.reload`):

   - in a different script `->` we get the reload
   - inside the script before the sandbox initialization `->` we get the reload
   - inside the script after the sandbox initialization `->` we get the reload
   - inside `nativeAdd()` `->` we get the reload

     //* still everything is allowed!!

7. experimentation with innerHTML:

   - in a different script after the sandbox `->` nothing
   - in a different script before the sandbox `->` nothing
   - in a different script inside a called function `->` we got the change
   - inside the script before the sandbox initialization `->` we get the change
   - inside the script after the sandbox initialization `->` we get the change
   - inside `nativeAdd()` `->` we get the change

   //* still nothing

8. i'll try to lock the window method:

   (a) `JSJS.SetLock(window,'locked');`

   //right after the initialization of the sandbox window.prompt(..); `->` we got the prompt

   (b) `jsObjs=JSJS.SetLock(window,'locked');` window.prompt(..); `->` we didn't get the prompt!! `->>>` good news //nothing!

(c) `jsObjs=JSJS.SetLock(document,'locked');` window.prompt(..); -> we didn't get the prompt -> bad news //nothing!!!

(d) `jsObjs=JSJS.SetLock(document,'unlocked');` window.prompt(..); -> we didn't get the prompt //nothing

(e) `jsObjs=JSJS.SetLock(document,'unlocked');`

//just before the shut down of the sandbox (after the prompt) window.prompt(..); -> we got the prompt

(f) `jsObjs=JSJS.SetLock(document,'unlocked');`

//right after the prompt and before the postMessages window.prompt(..); -> nothing

//* it stops when it reads SetLock for some reason

12/8/2014

30 mins

a break from js.js !!! -¿ hands on treehouse

- changed the default one (onclick event to true) -> nothing
- changed the `setPolicy` to `self.policy` of the other policies -> nothing

1 hour

LET'S GET BACK TO JSJS

- removed the 'create window.prompt' from the `wrapper.js` -> i could still get the prompt
- `JSJS.LockElement(window);` and try `window.prompt` -> i got the prompt
- `JSJS.LockElement(window.prompt);` and try `window.prompt` -> i got the prompt

13/8/2014

DID NOT INCLUDE TIME FOR SOME REASON

tried to manipulate the wrapper.js file

- cleared the whole content: no visibility of results
- deleted the `window.alert` method: still got the alert (which must have been expected since other methods that are not written in the file are allowed)
- changed the expected args of the `window.alert` to `null` and `objPtr`: still got the alert

--- to be honest, stuck, waiting for someone's reply (author's, supervisor's) to guide me what to do next..in the meantime writing on the thesis ---.

29/8/2014

DID NOT INCLUDE TIME FOR SOME REASON

- invoked the `JSJS.SetLock(window.alert)` function, and changed the content of this function inside the wrapper (added a `window.alert`) and we got the alert!

--- TRAVEL TO LONDON ---.

12/9/2014

6 hours to figure out how to print the time
1 min per load: (**30 mins**)

Page load experiments in sec (with TreeHouse):

Chrome: 0.069, 0.024, 0.024, 0.024, 0.027, 0.073, 0.026, 0.021, 0.022, 0.026 = 33.6 (mean time)
Firefox: 0.184, 0.253, 0.13, 0.121, 0.1, 0.153, 0.161, 0.124, 0.135, 0.161 = 152.2 (mean time)

IE: 0.088, 0.087, 0.097, 0.035, 0.049, 0.038, 0.034, 0.04, 0.054, 0.038 = 56.0 (mean time)

Page load experiments in ms(without): (**30mins**)

Chrome: 17, 16, 16, 14, 17, 13, 14, 14, 16, 16 = 15.3 (mean time)
Firefox: 40, 84, 34, 34, 36, 63, 62, 42, 39, 13 = 44.7 (mean time)
IE: 61, 12, 45, 36, 71, 37, 12, 10, 40, 10 = 33.4 (mean time)

13/9/2014

jsjs Latency: (**30 mins**)

Chrome:

NewRuntime: 14, 17, 14, 16, 16, 7, 15, 7, 16, 7 = 12.9
NewContext: 30, 36, 29, 30, 28, 29, 28, 32, 30, 25 = 29.7
GlobalClassInit: 8, 8, 8, 8, 9, 2, 9, 4, 9, 3 = 6.8
InitStandardClasses: 89, 86, 98, 95, 86, 44, 89, 86, 90, 81 = 84.4
Simple execution: 75, 76, 87, 74, 77, 11, 80, 6, 79, 7 = 57.2
DestroyContext: 33, 35, 36, 32, 36, 18, 34, 21, 36, 19 = 30
DestroyRuntime: 6, 3, 6, 4, 4, 3, 5, 2, 4, 4 = 4.1
Page load: 280, 283, 305, 273, 278, 137, 288, 179, 287, 168 = 247.8

Firefox:

NewRuntime: 19, 14, 14, 13, 19, 16, 14, 17, 14, 24 = 16.4
NewContext: 37, 31, 32, 31, 31, 30, 27, 37, 33, 46 = 33.5
GlobalClassInit: 23, 18, 20, 21, 21, 22, 22, 22, 24, 29 = 22.2
InitStandardClasses: 120, 123, 112, 105, 105, 115, 107, 170, 118, 156 = 102.5
Simple execution: 92, 86, 98, 102, 94, 97, 106, 107, 101, 137 = 102
DestroyContext: 47, 44, 52, 44, 47, 42, 52, 43, 57, 58 = 48.6
DestroyRuntime: 3, 4, 5, 8, 5, 7, 6, 4, 5, 6 = 5.3
Page load: 377, 341, 360, 349, 349, 354, 358, 431, 381, 488 = 378.8

IE:

NewRuntime: 41, 20, 14, 23, 20, 22, 23, 17, 20, 18 = 21.8
NewContext: 241, 140, 141, 133, 138, 124, 134, 127, 131, 124 = 143.3
GlobalClassInit: 7, 7, 7, 6, 9, 7, 7, 6, 7, 11 = 7.4
InitStandardClasses: 84, 101, 106, 90, 101, 91, 100, 94, 98, 94 = 95.9
Simple execution: 93, 106, 106, 121, 107, 90, 118, 105, 102, 98 = 104.6
DestroyContext: 39, 57, 44, 37, 39, 38, 66, 52, 52, 54 = 47.8
DestroyRuntime: 7, 10, 8, 10, 8, 6, 6, 9, 8, 8 = 8
Page load: 807, 464, 444, 434, 435, 396, 486, 430, 433, 423 = 475.2

Bibliography

[AGD05] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for javascript. In *ECOOP 2005-Object-Oriented Programming*, pages 428–452. Springer, 2005.

[AM98] Vinod Anupam and Alain Mayer. Security of web browser scripting languages: vulnerabilities, attacks, and remedies. In *Proceedings of the 7th USENIX Security Symposium*, pages 187–200, 1998.

[BJM09] Adam Barth, Collin Jackson, and John C Mitchell. Securing frame communication in browsers. *Communications of the ACM*, 52(6):83–91, 2009.

[BML+07] D Boneh, J Mitchell, R Ledesma, N Chou, and Y Teraguchi. Spoofguard, 2007.

[CC13] Moraila G. Shankaran A. Shi S. Warren A. M. Collberg C., Proebsting T. Measuring reproducibility in computer systems research. 2013.

[Cen] Mozilla Developer Center. Xpcnativewrapper.

[CF91] Robert Cartwright and Mike Fagan. Soft typing. In *ACM SIGPLAN Notices*, volume 26, pages 278–292. ACM, 1991.

[CHC08] Steven Crites, Francis Hsu, and Hao Chen. Omash: enabling secure web mashups via object abstractions. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 99–108. ACM, 2008.

[CHGL06] Richard S Cox, Jacob Gorm Hansen, Steven D Gribble, and Henry M Levy. A safety-oriented platform for web applications. In *Security and Privacy, 2006 IEEE Symposium on*. IEEE, 2006.

[Chi06] Eric Chien. Malicious yahooligans. *Virus Bulletin, August*, 2006.

[Cro08] Douglas Crockford. Adsafe: Making javascript safe for advertising, 2008.

[CVM+07] Stephen Chong, Krishnaprasad Vikram, Andrew C Myers, et al. Sif: Enforcing confidentiality and integrity in web applications. In *USENIX Security*, volume 7, 2007.

[DEHL08] John R Douceur, Jeremy Elson, Jon Howell, and Jacob R Lorch. Leveraging legacy code to deploy desktop applications on the web. In *OSDI*, volume 8, pages 339–354, 2008.

[DG09] Mohan Dhawan and Vinod Ganapathy. Analyzing information flow in javascript-based browser extensions. In *Computer Security Applications Conference, 2009. ACSAC'09. Annual*, pages 382–391. IEEE, 2009.

[DKBS+08] Frederik De Keukelaere, Sumeer Bhola, Michael Steiner, Suresh Chari, and Sachiko Yoshihama. Smash: secure component model for cross-domain mashups on unmodified browsers. In *Proceedings of the 17th international conference on World Wide Web*, pages 535–544. ACM, 2008.

[DLFMT04] Giuseppe A Di Lucca, Anna Rita Fasolino, M Mastoianni, and Porfirio Tramontana. Identifying cross site scripting vulnerabilities in web applications. In *Telecommunications Energy Conference, 2004. INTELEC 2004. 26th Annual International*, pages 71–80. IEEE, 2004.

[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*, pages 297–306. ACM, 2011.

[ELX07] Ulfar Erlingsson, V Benjamin Livshits, and Yinglian Xie. End-to-end web application security. In *HotOS*, 2007.

[FC08] Bryan Ford and Russ Cox. Vx32: Lightweight user-level sandboxing on the x86. In *USENIX Annual Technical Conference*, pages 293–306. Boston, MA, 2008.

[FHEW08] Adrienne Felt, Pieter Hooimeijer, David Evans, and Westley Weimer. Talking to strangers without taking their candy: isolating proxied content. In *Proceedings of the 1st Workshop on Social Network Systems*, pages 25–30. ACM, 2008.

[Fla02] David Flanagan. *JavaScript: the definitive guide*. O'Reilly Media, Inc., 2002.

[FWB10] Matthew Finifter, Joel Weinberger, and Adam Barth. Preventing capability leaks in secure javascript subsets. In *NDSS*, 2010.

[GL09] Salvatore Guarnieri and V Benjamin Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code. In *USENIX Security Symposium*, pages 151–168, 2009.

[GPR04] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *NDSS*, 2004.

[Gre] E. Grey. Jsandbox. `https://github.com/eligrey/jsandbox`.

[GTK08] Chris Grier, Shuo Tang, and Samuel T King. Secure web browsing with the op web browser. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 402–416. IEEE, 2008.

[GWS11] Bernd Grobauer, Tobias Walloschek, and Elmar Stocker. Understanding cloud computing vulnerabilities. *Security & privacy, IEEE*, 9(2):50–57, 2011.

[HV05] Oystein Hallaraker and Giovanni Vigna. Detecting malicious javascript code in mozilla. In *Engineering of Complex Computer Systems, 2005. ICECCS 2005. Proceedings. 10th IEEE International Conference on*, pages 85–94. IEEE, 2005.

[IM09] Scott Isaacs and Dragos Manolescu. Websandbox-microsoft live labs, 2009.

[Ing] L. Ingram. Lawnsee github. `https://github.com/lawnsea/TreeHouse`.

[IW12] Lon Ingram and Michael Walfish. TreeHouse: JavaScript sandboxes to help web developers help themselves. In *USENIX ATC*, 2012.

[jT] jQuery Team. jquery. `http://jquery.com/`.

[JW07] Collin Jackson and Helen J Wang. Subspace: secure cross-domain communication for web mashups. In *Proceedings of the 16th international conference on World Wide Web*, pages 611–620. ACM, 2007.

[KL10] Emre Kiciman and Benjamin Livshits. Ajaxscope: A platform for remotely monitoring the client-side behavior of web 2.0 applications. *ACM Transactions on the Web (TWEB)*, 4(4):13, 2010.

[KSW+13] Vineeth Kashyap, John Sarracino, John Wagner, Ben Wiedermann, and Ben Hardekopf. Type refinement for static analysis of javascript. In *Proceedings of the 9th symposium on Dynamic languages*, pages 17–26. ACM, 2013.

[MD11] James Mickens and Mohan Dhawan. Atlantis: robust, extensible execution environments for web applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 217–231. ACM, 2011.

[MEH10] James W Mickens, Jeremy Elson, and Jon Howell. Mugshot: Deterministic capture and replay for javascript applications. In *NSDI*, volume 10, pages 159–174, 2010.

[ML10] Leo A Meyerovich and Benjamin Livshits. Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 481–496. IEEE, 2010.

[MMT09] Sergio Maffeis, John C Mitchell, and Ankur Taly. Isolating javascript with filters, rewriting, and wrappers. In *Computer Security–ESORICS 2009*, pages 505–522. Springer, 2009.

[MMT10] Sergio Maffeis, John C Mitchell, and Ankur Taly. Object capabilities and isolation of untrusted web applications. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 125–140. IEEE, 2010.

[MSL+08] Mark S Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Safe active content in sanitized javascript. Technical report, Technical report, Tech. Rep., Google, Inc, 2008.

[MT09] Sergio Maffeis and Ankur Taly. Language-based isolation of untrusted javascript. In *Computer Security Foundations Symposium, 2009. CSF'09. 22nd IEEE*, pages 77–91. IEEE, 2009.

[Pil] M Pilgrim. Greasemonkey for secure data over insecure networks/sites, july 2005.

[Pro] Open Web Application Security Project. The ten most critical web application security vulnerabilities. `http://umn.dl.sourceforge.net/sourceforge/owasp/OWASPTopTen2004.pdf`.

[PS01] Vassilis Prevelakis and Diomidis Spinellis. Sandboxing applications. In *USENIX Annual Technical Conference, FREENIX Track*, pages 119–126, 2001.

[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*, pages 47–60. ACM, 2009.

[RBP09] Charles Reis, Adam Barth, and Carlos Pizano. Browser security: lessons from google chrome. *Queue*, 7(5):3, 2009.

[RDW+07] Charles Reis, John Dunagan, Helen J Wang, Opher Dubrovsky, and Saher Esmeir. Browsershield: Vulnerability-driven filtering of dynamic html. *ACM Transactions on the Web (TWEB)*, 1(3):11, 2007.

[RG09] Charles Reis and Steven D Gribble. Isolating web programs in modern browser architectures. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 219–232. ACM, 2009.

[saW] The samy worm. `http://namb.la/popular`.

[Sec] Secunia. Microsoft internet explorer window injection vulnerability. `http://secunia.com/advisories/13251/`.

[SGL04] Stefan Saroiu, Steven D Gribble, and Henry M Levy. Measurement and analysis of spyware in a university environment. In *NSDI*, pages 141–153, 2004.

[SS98] Christopher Small and Margo I Seltzer. Misfit: Constructing safe extensible systems. *Concurrency, IEEE*, 6(3):34–41, 1998.

[TBK12] Jeff Terrace, Stephen R Beard, and Naga Praveen Kumar Katta. Javascript in javascript (js. js): Sandboxing third-party scripts. *USENIX WebApps*, 2012.

[teaa] Dojo team. Dojo toolkit. `http://dojotoolkit.org/`.

[Teab] Ext JS Team. Ext js. `http://www.sencha.com/products/extjs`.

[Teac] Google Caja Team. Google-caja: A source-to-source translator for securing javascript-based web.

[Tead] Prototype Team. Prototype. `http://www.prototypejs.org/`.

[Teae] Twitter Team. Twitter widgets. `https://twitter.com/about/resources/widgets`.

[Teaf] YUI Team. Yui. `http://yuilibrary.com/`.

[Teag] Zepto.js Team. Zepto.js. `http://zeptojs.com/`.

[The] P. Theriault. Bawks. `http://bawks.creativemisuse.com/`.

[Thi05] Peter Thiemann. Towards a type system for analyzing javascript programs. In *Programming Languages and Systems*, pages 408–422. Springer, 2005.

[TLGV10] Mike Ter Louw, Karthik Thotta Ganesh, and VN Venkatakrishnan. Adjail: Practical enforcement of confidentiality and integrity policies on web advertisements. In *USENIX Security Symposium*, pages 371–388, 2010.

[TMK10] Shuo Tang, Haohui Mai, and Samuel T King. Trust and protection in the illinois browser operating system (ibos). In *OSDI*, pages 17–32, 2010.

[WFHJ07] Helen J Wang, Xiaofeng Fan, Jon Howell, and Collin Jackson. Protection and communication abstractions for web browsers in mashupos. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 1–16. ACM, 2007.

[WGM+09] Helen J Wang, Chris Grier, Alexander Moshchuk, Samuel T King, Piali Choudhury, and Herman Venter. The multi-principal os construction of the gazelle web browser. In *USENIX Security Symposium*, volume 28, 2009.

[Wil05] S Willison. Understanding the greasemonkey vulnerability, 2005.

[WLAG94] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. In *ACM SIGOPS Operating Systems Review*, volume 27, pages 203–216. ACM, 1994.

[YCIS07] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. Javascript instrumentation for browser security. In *ACM SIGPLAN Notices*, volume 42, pages 237–249. ACM, 2007.

[YNKM09] Alexander Yip, Neha Narula, Maxwell Krohn, and Robert Morris. Privacy-preserving browser-side scripting with bflow. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 233–246. ACM, 2009.

[YSD+09] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable,

untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 79–93. IEEE, 2009.

[Zei] A. Zeigler. Internet explorer 8 and loosely-coupled ie. `http://blogs.msdn.com/b/ie/archive/2008/03/11/ie8-and-loosely-coupled-ie-lcie.aspx`.

[ZYG08] Saman Zarandioon, Danfeng Yao, and Vinod Ganapathy. Omos: A framework for secure communication in mashup applications. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*, pages 355–364. IEEE, 2008.