POLITECNICO DI MILANO
Scuola di Ingegneria dell'Informazione
Corso di Laurea in Ingegneria Informatica

# Evolution of Client-side Web Attacks and Defences against Malicious JavaScript code: A Retrospective, Systematisation Study

**Relatore:**
Prof. Stefano ZANERO
**Correlatore:**
Dr. Federico MAGGI

**Tesi di Laurea Magistrale di:**
Yin Zhining, matr. 766473

Anno Accademico 2012–2013

# Abstract

JavaScript is the most common scripting language that is used to build rich web interfaces. It has also been abused to harm customers' machines, such as stealing customers' private data, for a long time. JavaScript is a prototype-based scripting language that is dynamic, weakly-typed and has first-class functions. Because of these peculiarities, it is very difficult to apply traditional program-analysis techniques such as static analysis and dynamic analysis. Indeed, the research in this field is very active, motivated by the need for tools that can recognize malicious JavaScript code (e.g., code that attempt to exploit a browser vulnerability) and protect clients (e.g, browsers) from being compromised by malicious JavaScript code.

Unfortunately, there is no paper that systematise the knowledge in this field. The goal of this thesis is to systematise the research approaches published in the proceedings of top computer security conferences between 2005 and 2012. In addition, we attempt to draw future trends of JavaScript-based threats and mitigation approaches. We study the evolution of both JavaScript-based threats and mitigation approaches and propose a taxonomy. We describe 34 papers in detail and compare the advantages and disadvantages among them.

# Sommario

**Il pericolo crescente di codice JavaScript malevolo.** Per migliorare la visualizzazione dei siti web sul lato cliente, JavaScript é il linguaggio piú comunemente scelto dagli sviluppatori web, ed é supportato da tutti i browser web come Google Chrome, Firefox e Internet Explorer. Il linguaggio JavaScript [Flanagan, 1998] é stato sviluppato da Netscape come linguaggio di scripting leggero con funzionalitá orientate agli oggetti ed é stato successivamente standardizzato da ECMA [ECMA, 2011].

JavaScript puó essere abusato per scopi malevoli (ad esempio, sfruttare le vulnerabilitá dei browser per causare la perdita di dati privati). JavaScript ha le seguenti quattro caratteristiche: basato su prototipo, dinamica, debolmente tipizzato ed espone funzioni di prima classe che rendono molto difficile da rilevare le minacce basate su di esso. Al giorno d'oggi, l'abuso maligno di Javascript é diventata la principale minaccia per i clienti. Secondo un rapporto pubblicato da Sophos Labs indicano che il numero di frammenti JavaScript maligni analizzati ogni giorno nel 2010 - circa 95.000 campioni - quasi raddoppiato dal 2009 citep 2011. Ci sono diversi tipi di minacce basate su JavaScript contro i browser attuali, come ad esempio:

1. Attacchi drive-by download, che significa una vulnerabilitá nel browser web o uno dei suoi componenti / estensioni (ad esempio, Acrobat Reader o Flash plug-in) viene sfruttata per imporre ai client di scaricare ed eseguire codice arbitrario.

2. Cross-Site Scripting (XSS) che consente ad un utente malintenzionato di iniettare codice JavaScript malevolo in pagine web.

3. Attacchi heap spraying, che sfruttano vulnerabilitá di heap/buffer overflow del browser o browser dei suoi plug-ins per allocare una grande quantitá di memoria heap, di solito utilizzando array di stringhe che contengono shell-code e NOP sleds.

**Lavori precedenti.** Motivata dalla necessitá di proteggere i clienti dalle minacce basate su JavaScript, la ricerca in questo campo é molto attiva. Diversi, nuovi approcci e strumenti sono stati proposti, ad esempio: Wepawet [Cova et al., 2010] esegue un'analisi on-line di una determinata pagina, al fine di rilevare gli attacchi drive-by download. Cujo [Rieck et al., 2010] esegue l'analisi online, ma introduce un overhead di piú di 1,5 secondi su siti JavaScript pesanti come Facebook, che incide negativamente sulla qualitá dell'esperienza dell'utente. Gatekeeper [Guarnieri and Livshits, 2009] e Google Caja [Mark S. Miller, 2007] tentano di trovare un modo per eseguire codice JavaScript arbitrario in un ambiente protetto, il che richiede di lavorare su un sottoinsieme della specifica JavaScript completa, ad esempio, Gatekeeper rimuove costrutti del linguaggio come `eval()` e `document.write()` dalla specifica JavaScript per la loro analisi.

**La motivazione di questa tesi.** Finora, ci sono numerosi tipi di minacce basate su JavaScript e proposti approcci che ne mitigano gli effetti. Ma, purtroppo, non c'é un tesi che istematizza

la conoscenza in questo campo. Questa tesi é la prima tesi che sistematizza la conoscenza della sicurezza JavaScript. Pensiamo che questa tesi puó aiutare le persone che e nuovo in questo campo, nella prospettiva di rispondere alle seguenti problematiche:

1. Quali minacce basate su JavaScript e approcci di mitigazione sono stati sviluppati.
2. Per attenuare uno specifico minaccia basato su JavaScript, quali tipi di contromisure sono efficaci per fermarla.
3. Quale puó essere la prossima minaccia basata su JavaScript e quali contromisure possono essere preparate per mitigarla.

**Principali contributi.** Per riassumere, questa tesi presenta i seguenti contributi:

1. Studiamo l'evoluzione di entrambe le minacce basate su JavaScript e le tecniche di mitigazione contro di loro. Esponendole in ordine della loro evoluzione temporale.
2. Forniamo una tassonomia per classificare le minacce basate su JavaScript esistenti e gli approcci di mitigazione e di descrivere le differenze, i vantaggi, gli svantaggi delle diverse classi di approcci di mitigazione.
3. Usiamo due tavoli di correlare le tecniche di mitigazione con le minacce basate su JavaScript per mostrare l'efficacia delle tecniche di mitigazione.
4. Descriviamo i documenti cui si fa riferimento in dettaglio come prova per la conoscenza che abbiamo di sistematizzare.
5. Indichiamo i prossimi possibili attacchi.

# Acknowledgement

My deepest gratitude goes to Prof. Stefano Zanero and Dr. Federico Maggi who supervised me during the whole process of writing this thesis. They taught me a lot including the scientific attitude to any works. I feel extremely happy to work with them and learned a lot from them.

I want to thank my parents. Without them, i will not even have this opportunity to study in Politecnico di Milano, and work with the best professors and students. I hope they have good health and have a happy life in Shanghai.

I also want to thank all of my family members who always support me and take care of my parents for me.

# List of Figures

2.1 Evolution of JavaScript-based threats . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
2.2 Number of papers that target different **vectors** in each year. . . . . . . . . . . . . . . . 15
2.3 Time-line of mitigation approaches against JavaScript-based threats. . . . . . . . . . 15
2.4 Number of papers that mitigate JavaScript-based threats in each year. . . . . . . . . . 16
2.5 Taxonomy of mitigation approaches against JavaScript-based threats. . . . . . . . . . 17
2.6 An example of AST. Source: `http://en.wikipedia.org/wiki/Abstract_syntax_tree` . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
2.7 Example of JavaScript program and its points-to graphs. Top: program code, bottom left: conventional graph, bottom right: graph with properties. Source: [Jang and Choe, 2009] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19

3.1 System overview of ADSandbox. Source: [Dewald et al., 2010]. . . . . . . . . . . . 28
3.2 Spectator architecture. Source: [Livshits and Cui, 2008]. . . . . . . . . . . . . . . . . 29
3.3 ZOZZLE training process. Source: [Curtsinger et al., 2011]. . . . . . . . . . . . . . . 30
3.4 Deployment of BrowserShield. Source: [Reis et al., 2006]. . . . . . . . . . . . . . . . 31
3.5 HTML and script translation of BrowserShield. Source: [Reis et al., 2006]. . . . . . 31
3.6 De-obfuscation process. Source: [Lu and Debray, 2012]. . . . . . . . . . . . . . . . . 35
3.7 The overall analysis process of VEX. Source: [Bandhakavi et al., 2010]. . . . . . . . 38
3.8 js.js architecture for example application. Source: [Terrace et al., 2012]. . . . . . . . 39
3.9 GATEKEEPER deployment. Source: [Guarnieri and Livshits, 2009]. . . . . . . . . . 40
3.10 GATEKEEPER analysis architecture. Source: [Guarnieri and Livshits, 2009]. . . . . 41
3.11 Schematic description of Cujo. Source: [Rieck et al., 2010]. . . . . . . . . . . . . . . 48
3.12 Example of reports generated by Cujo. Up: Report of static analysis. Bottom: Report of dynamic analysis. Source: [Rieck et al., 2010]. . . . . . . . . . . . . . . . . . 48
3.13 Architecture of PJScan. Source: [Laskov and Srndic, 2011]. . . . . . . . . . . . . . . 49
3.14 Tokens' definition in PJscan. Source: [Laskov and Srndic, 2011]. . . . . . . . . . . . 50
3.15 Overall architecture of MDscan. Source: [Tzermias et al., 2011]. . . . . . . . . . . . 50
3.16 NOZZLE system architecture. Source: [Ratanaworabhan et al., 2009]. . . . . . . . . 51

# List of Tables

2.1 Comparison between static analysis and dynamic analysis. . . . . . . . . . . . . . . . 21
2.2 Papers the propose mitigation techniques against **vectors** and **evasions**. . . . . . . 24
2.3 Papers the propose mitigation techniques against goals, **techniques** and **evasions**. 25

3.1 Initial source of tainted values. Source: [Vogt et al., 2007]. . . . . . . . . . . . . . . . . 33
3.2 Sensitive sources of Sabre. Source: [Dhawan and Ganapathy, 2009]. . . . . . . . . . 37
3.3 Insensitive sinks of Sabre. Source: [Dhawan and Ganapathy, 2009]. . . . . . . . . . 37

# Contents

**1 Introduction** **8**

**2 Systematisation of the research approaches** **10**
2.1 Brief introduction on JavaScript . . . . . 10
2.2 Referenced material . . . . . 11
2.3 Evolution of JavaScript-based threats and defences . . . . . 11
2.3.1 JavaScript-based threats . . . . . 12
2.3.2 Mitigation techniques . . . . . 14
2.4 Taxonomy of defence techniques against JavaScript-based threats . . . . . 16
2.4.1 Global taxonomy . . . . . 16
2.4.2 Detection techniques . . . . . 17
2.4.3 Prevention techniques . . . . . 22
2.5 Correlation between threats and mitigation techniques . . . . . 22

**3 Detailed survey and comparison of research approaches** **26**
3.1 Malicious JavaScript . . . . . 26
3.2 XSS vulnerabilities . . . . . 31
3.3 Environment matching . . . . . 33
3.4 Obfuscation . . . . . 34
3.5 JavaScript extensions . . . . . 36
3.6 Mash-ups . . . . . 38
3.7 Drive-by download . . . . . 44
3.8 PDFs with embedded JavaScript . . . . . 49
3.9 Heap spraying . . . . . 51

**4 Conclusions** **53**
4.1 Future research directions . . . . . 53
4.2 Conclusions . . . . . 54

# Chapter 1

# Introduction

**The increasing danger of malicious JavaScript code.** To enhance the display of web sites on the client side, JavaScript is the most common language that is chosen by web developers, and it is supported by all the web browsers such as Google Chrome, Firefox and Internet Explorer. JavaScript language [Flanagan, 1998] was developed by Netscape as a lightweight scripting language with object-oriented capabilities and was later standardized by ECMA [ECMA, 2011].

JavaScript can be abused for malicious purpose (e.g, exploit the vulnerabilities of browser to cause private data leakage). JavaScript has the following four peculiarities: prototype-based, dynamic, weakly-typed and first-class functions, which make it very difficult to detect JavaScript-based threats. Nowadays, abusing JavaScript maliciously has become the major threat to clients. According to a report from Sophos Labs indicates that the number of JavaScript malware pieces analysed by Sophos Labs every day in 2010 - about 95,000 samples - nearly doubled from 2009 [Labs, 2011]. There are many different kinds of JavaScript-based threats against current browsers, such as:

1. Drive-by download attacks, which means a vulnerability in the web browser or one of its components/extensions (e.g., Acrobat Reader or Flash plug-ins) is exploited to force clients to download and execute arbitrary code.
2. Cross-Site Scripting (XSS) vulnerabilities that enable an attacker to inject malicious JavaScript code into web pages.
3. Heap spraying attacks, which exploit heap/buffer overflow vulnerabilities of the browser or installed browser-plug-ins by allocating big amount of memory on the heap, usually utilising arrays of strings that contain shell-code and NOP sleds.

**Previous Works.** Motivated by the need to protect clients from JavaScript-based threats, the research in this field is very active. Different novel approaches and tools were proposed, for example: Wepawet [Cova et al., 2010] performs an on-line analysis of a given page in order to detect drive-by download attacks. Cujo [Rieck et al., 2010] performs on-line analysis, but introduces an overhead of more than 1.5 seconds on JavaScript-heavy sites such as Facebook, which negatively impacts the user experience. Gatekeeper [Guarnieri and Livshits, 2009] and Google Caja [Mark S. Miller, 2007] attempt to find a way to execute arbitrary JavaScript in a secure environment, which requires working on a subset of the complete JavaScript specification, e.g., Gatekeeper removes language constructs such as `eval()` and `document.write()` from the JavaScript specification for their analysis.

**Motivation of this thesis.** Until now, there are numerous kinds of JavaScript-based threats and mitigation approaches against them. But unfortunately, there is no paper that systematises the knowledge in this field. This thesis systematises the knowledge of JavaScript security for the first time. We think this thesis can help people who is new to this field with a view to answer the following important questions:

1. What JavaScript-based threats and mitigation approaches have been developed.
2. To mitigate a specific JavaScript-based threat, what kind of mitigation approach is effective.
3. What can be the next JavaScript-based threat and what countermeasures can be prepared to mitigate it.

**Main Contribution.** To summarise, this thesis presents the following contributions:

1. We study the evolution of both JavaScript-based threats and mitigation techniques against them. We draw two time-lines to illustrate the evolution.
2. We provide a taxonomy to classify the existing JavaScript-based threats and mitigation approaches and describe the differences, advantages, disadvantages of different classes of mitigation approaches.
3. We use two tables to correlate mitigation techniques with JavaScript-based threats to show the effectiveness of mitigation techniques.
4. We describe the referenced papers in detail as proof to the knowledge that we systematise.
5. We indicate the next possible attacks.

**Organization of this thesis.** The remainder of this thesis is structured as follows:

1. In Chapter 2, we give a brief introduction on JavaScript language. We systematise the knowledge that we learned from all referenced papers including the evolution of JavaScript-based threats and mitigation approaches, classification of mitigation approaches, and correlation between JavaScript-based threats and mitigation approaches.
2. In Chapter 3, we describe 34 papers in detail to prove the knowledge described in Chapter 3. We compare the difference, advantages and disadvantages among all the approaches.
3. In Chapter 4, we draw some conclusions. We indicate the next possible attacks and suggest the possible ways to mitigate next attacks.

# Chapter 2

# Systematisation of the research approaches

In this chapter, to describe the knowledge that we systematise from referenced papers in an organized way, we divide this chapter into 4 sections. First, we start from a brief introduction on JavaScript language, which explains why and how JavaScript can be abused. Second, we use two time-lines to present the evolution of JavaScript-based threats and mitigation techniques. Third, we build a classification for mitigation techniques. Finally we present two tables to illustrate what class of mitigation techniques is effective against what JavaScript-based threats.

## 2.1 Brief introduction on JavaScript

JavaScript is the most common scripting language that is used to build dynamic websites. Also the situation of abusing JavaScript is severe, since JavaScript has the following peculiarities:

- Prototype-based scripting language

  JavaScript is an object-oriented language that uses prototype inheritance. Prototype inheritance is a form of object-oriented code reuse. In the prototype inheritance form, objects inherit directly from other objects. In JavaScript, every JavaScript object has a secret link to another JavaScript object, which created it, forming a chain. When a JavaScript object is asked for a property that it does not have, its parent object is asked continually up the chain until the property is found or until the root object is reached. In JavaScript, root objects such as `Cache`, `Document` are pre-defined.

  This peculiarity provides attackers with a way to illegally access the root objects through the prototype inheritance chain. Since the root objects are global object, if they are modified for malicious purposes, the modification will affect all other objects.

- Dynamic

  JavaScript is a dynamic language since it allows JavaScript code to be dynamically generated at run-time(e.g., code obfuscation). Code obfuscation is a technique to transform source code to e.g., protect code copyright.

  This peculiarity also makes static analysis very difficult, because the code is dynamically generated and can only be analysed at run-time.

- Weakly-typed

JavaScript is a weakly typed scripting language. Compared to strong typed language, JavaScript removes some rules during compilation (e.g., JavaScript allows overloading and type conversion). Weakly typing does create more runtime errors and exceptions, such as heap overflow, which can be used to inject and execute arbitrary malicious code.

- First-class functions

  This peculiarity means that JavaScript allows passing functions as arguments to other functions, returning them as the values from other functions, and assigning them to variables or storing them in data structures. This increase the difficulty to apply static analysis.

## 2.2 Referenced material

To ensure the knowledge that is systematised in this thesis and our final conclusions are complete, we have tried our best to collect as many related papers as possible. The collecting criteria is composed by 2 steps: (1) We select papers that are related to JavaScript security from the proceedings of the top conferences in computer security since 2005:

- IEEE S&P (IEEE Symposium on Security and Privacy)
- CCS (ACM Conference on Computer and Communications Security)
- NDSS (ISOC Network and Distributed System Security Symposium)
- USENIX (Usenix Security Symposium)
- ACSAC (Annual Computer Security Applications Conference)
- RAID (International Symposium on Recent Advances in Intrusion Detection)

(2) After reading the papers that are selected in step 1, we also read all their reference papers. If there are new related papers found, we continue repeating step 2 until no more new related papers are found. As a result, we choose 88 related papers, among which 34 papers propose mitigation approaches against JavaScript-based threats. From these 34 papers, we

- observed the evolution and the focus of the research approaches, and
- inferred the evolution of the JavaScript-based threats.

The rest of the papers are important to understand a specific technique; they can be considered as secondary references.

## 2.3 Evolution of JavaScript-based threats and defences

In this section, we present two time-lines, which summarise the evolution of, respectively, JavaScript-based threats and the research approaches proposed to mitigate or defeat them. We notice immediately the well-known arms race. On one hand, the evolution trend of JavaScript-based threats focuses on developing new **vectors** , new **techniques** or new **evasions** to evade existing mitigation approaches. On the other hand, the mitigation approaches also evolve, the efficiency and coverage of threats are both improved through time.

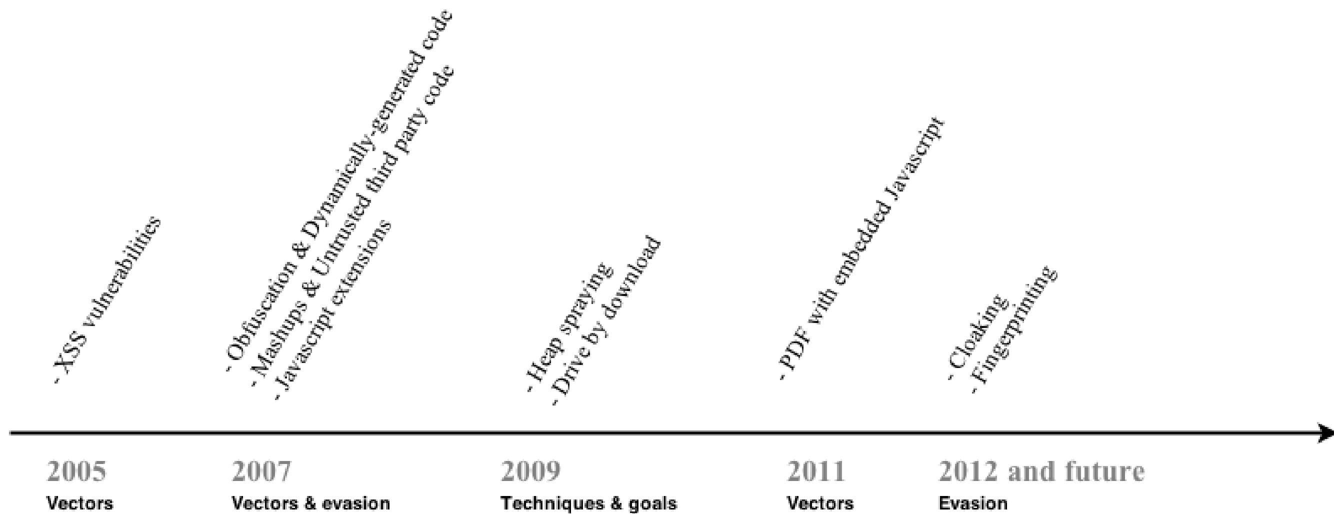


Figure 2.1: Evolution of JavaScript-based threats

### 2.3.1 JavaScript-based threats

The time-line in Fig. 2.1 shows the evolution of JavaScript-based threats and their characteristics. We associate to each year one of the following semantic keywords: **vectors**, **evasions**, and **technique**. These keywords characterise JavaScript-based threats from different angles. For example, **vectors** are the ways to inject and execute malicious JavaScript code, **evasions** means the techniques to hide the source malicious JavaScript code and evade known mitigation approaches, **technique** means the technique that is used to write malicious JavaScript code to exploit browser/browser plug-ins vulnerabilities. We tag each year with at least one keywords that better indicate the focus of the researches on that year.

Before explaining all these keywords in detail, we want to mention that there is a common goal behind any JavaScript-based threats that is writing malicious JavaScript code to exploit browser vulnerabilities and making the code executed. Drive-by download is the most common subgoal to pursue this common goal. In our thesis, drive-by download regards executing malicious JavaScript code to force clients to download computer malware (e.g., malicious JavaScript code, binary malware) unintentionally and execute it. We can see from Fig. 2.1 that the researches started to focus on drive-by download around 2009. However, drive-by download is still common today, although it is more complex today than it was in 2009.

- **Vectors**

  **Vectors** are the means to implement an attack, thus, in the context of this thesis, the ways to inject malicious JavaScript code and execute it. In our time-line, the referenced papers focus on mitigating 4 **vectors**:

  XSS vulnerabilities

  XSS is a well-known vulnerability found in web applications `https://www.owasp.org/index.php/Top_10_2013`, which can be used to inject arbitrary code into a webpage. Unsurprisingly, cybercriminals have been exploiting XSS vulnerabilities to

inject malicious JavaScript code, which ends up to be executed on the visitors' machines.

Mash-ups

A mash-up is a web page that uses and combines data, JavaScript, presentation or functionality from two or more sources (e.g., JavaScript advertising). This practice may create data-leak vulnerabilities, which means leaking private data from one source to another (e.g., from safe, trusted source to unsafe, untrusted source/third party). This provides a way for attackers to load and execute malicious code on the client side in the form of third party code. When a customer visits a mash-up that includes a piece of malicious JavaScript code, the browser automatically invokes the malicious third-party code and executes it.

JavaScript-based browser extensions

This **vector** may also create data-leak vulnerabilities like mash-ups. In this case, the malicious JavaScript code is injected inside browser extensions. When the customer visits a webpage that requires the execution of malicious browser extension, the malicious code gets executed.

PDFs with embedded JavaScript

Differently from previous **vectors**, which target the execution of JavaScript code in a browser and the exploitation of the browser environment. In this case, the malicious JavaScript code is injected into PDF files and gets executed only with the opening of PDF file (e.g., within a PDF reader, or browser plug-in for rendering PDF).

- **Evasion**

  **Evasion** means the techniques to evade the existing, known detection or prevention mechanisms, making malicious code hard to be detected. While reading the research works since 2005, we notice the following **evasion** techniques:

Obfuscation, dynamically generated code

Obfuscation is a method to transform source code such that it is harder to read and understand. Obfuscation is used to legitimately protect code copyright, but it can also be abused to hide the code that is responsible for attacks to make static analysis harder. In JavaScript, it can be simply achieved by using the `eval()` function. In general, the dynamic features of JavaScript allow other ways to dynamically generate code, for example, by invoking initial pieces of JavaScript code from different sources, and compose a final malicious JavaScript code at run-time. This also increases the difficulty for static analysis.

Environment matching

If attackers can detect the environment of the client (e.g., vulnerabilities, browser version, OS), then attackers can deliver tailored malicious JavaScript code. Or, if the client is actually a sandbox that runs JavaScript code in a controlled environment to detect whether it is malicious, then attacker can decide not to reveal the true malicious behaviour of the code.

This kind of **evasion** now starts to be widely used. From our referenced papers we found 2 techniques that are specifically used for this purpose:

  – Fingerprinting

Browser fingerprinting is a technique in which a variety of environment variables are evaluated to assess the capabilities of the browser. Privacy advocates show that browser fingerprinting can be used to track users across sessions without the help of cookies as browsers carry unique information that results in unique fingerprints.

– Cloaking

Cloaking can be divided into server-side cloaking and client-side cloaking. In server-side cloaking the server can choose not to deliver the malicious code (e.g., when the request comes from a suspicious IP, for instance, the IP of a research laboratory). Meanwhile client-side cloaking is achieved by using JavaScript to identify client-side characteristics (e.g., malware that checks if images have been successfully loaded before executing its attack.).

- **Technique**

  This **vector** refers to the techniques that are used to exploit the clients' vulnerabilities to execute machine code.

  – Heap spraying

  Heap spraying is a technique to exploit memory errors such as heap overflow to execute arbitrary code. Heap overflow is a type of buffer overflow that occurs in the heap data area. Memory on the heap is dynamically allocated by the application at runtime and typically contains program data. Exploitation is performed by corrupting this data in specific ways to cause the application to overwrite internal structures such as linked list pointers. The canonical heap overflow technique overwrites dynamic memory allocation linkage and uses the resulting pointer exchange to overwrite a program function pointer. In a practice of implementing heap spraying, code that sprays the heap attempts to put a certain sequence of bytes at a predetermined location in the memory of a target process by having it allocate large blocks on the process' heap and fill the bytes in these blocks with the right values.

  In JavaScript, heap spraying can be implemented by allocating large strings. The most common technique used is to start with a string of one character and concatenating it with itself over and over. This way, the length of the string can grow exponentially up to the maximum length allowed by the scripting engine. Depending on how the browser implements strings, the heap spraying code makes copies of the long string with shell code and stores these in an array, up to the point where enough memory has been sprayed to ensure the exploit works.

**Conclusion** To summarise, Fig. 2.2 is a bar-chart that shows the number of papers that mitigate different **vectors** in each year. We find that the focus of attacks evolve from malicious code injection through XSS vulnerabilities to source code transformation, **evasion** of mitigation approaches and the developing of new **techniques** to exploit new vulnerabilities, which all of them make defending more and more difficult.

### 2.3.2 Mitigation techniques

Fig. 2.3 shows a time-line that presents the evolution of mitigation approaches against JavaScript-based threats. The keywords in this time-line are explained in subsection 2.3.1. Here, we describe some observations we find from this time-line:

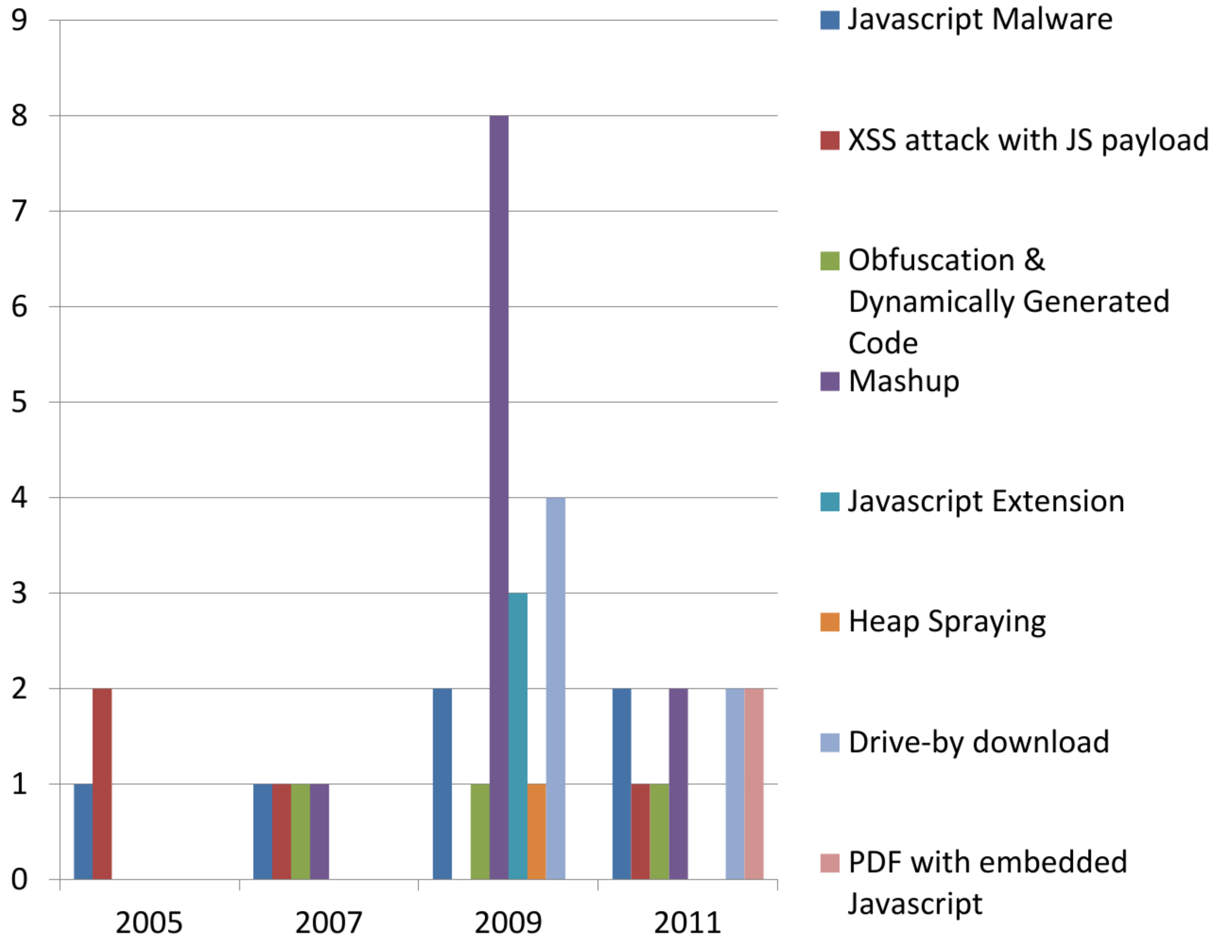


Figure 2.2: Number of papers that target different **vectors** in each year.

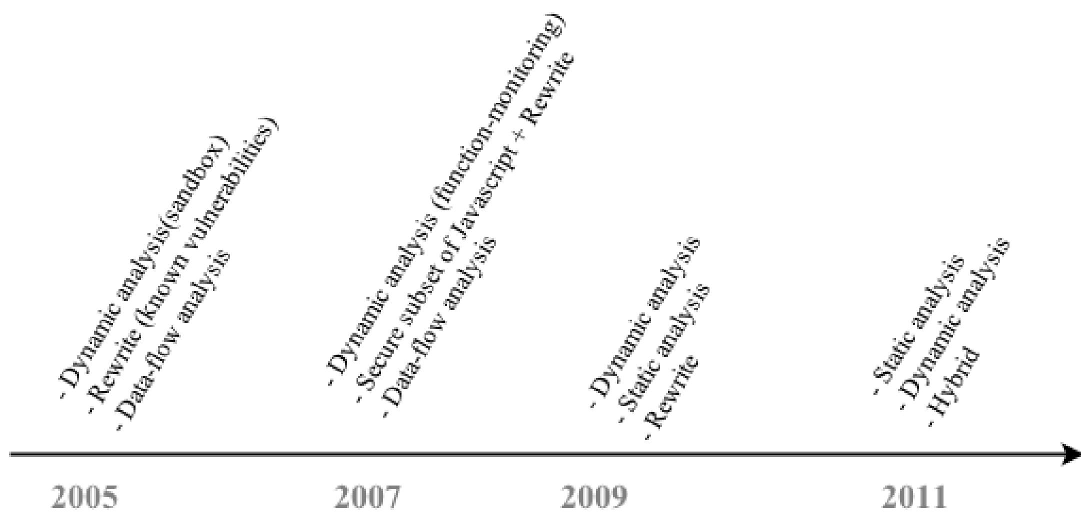


Figure 2.3: Time-line of mitigation approaches against JavaScript-based threats.

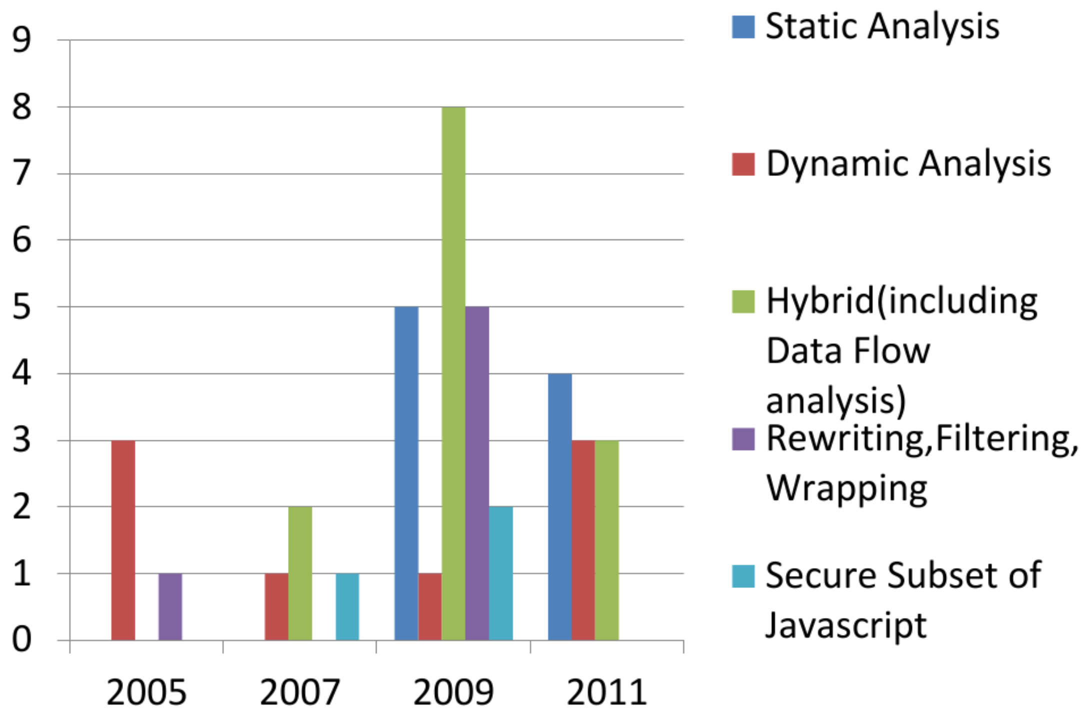


Figure 2.4: Number of papers that mitigate JavaScript-based threats in each year.

- Dynamic analysis can be used to restrict the privileges by running JavaScript in sandbox (e.g., by restricting the access of some private data in the browser). Dynamic analysis can also be used as a procedure of logging code behaviour (e.g, environment changing, reading browser cookie) during code execution to determine whether a piece of JavaScript code is malicious or not.

- Rewriting techniques can be applied to rewrite known malicious JavaScript code or to reduce code to a pre-defined secure subset of JavaScript.

- Static analysis compensates for the limitations of dynamic analysis. Static analysis is performed without code execution, and is performed directly on source or the object code. Hybrid analysis combines static and dynamic analysis. The idea is to use both the advantages of static analysis and dynamic analysis and evade their limitations.

We also provide another bar-chart to show the trend of the usage of mitigation techniques in Fig. 2.4

## 2.4 Taxonomy of defence techniques against JavaScript-based threats

In this section, we describe our classification of mitigation techniques. We start from presenting a global taxonomy, then explain two main classes of mitigation techniques in detail including the comparison among all techniques inside each class.

### 2.4.1 Global taxonomy

Fig. 2.5 shows the global taxonomy of the research approaches to mitigate or defeat JavaScript-based threats, which we divide into **Detection** and **Prevention** approaches. **Detection** approaches

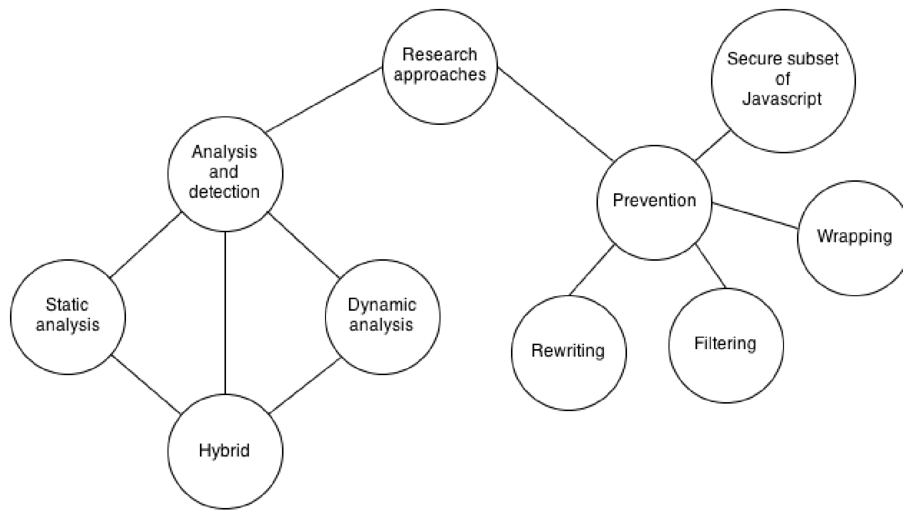


Figure 2.5: Taxonomy of mitigation approaches against JavaScript-based threats.

aim at detecting malicious JavaScript code by analysing JavaScript code. Usually new detection techniques are developed after new attacks are discovered, since it requires time to capture the characteristics of new attacks. **Prevention** approaches propose mechanisms that can stop attacks before damage is taken without analysing JavaScript code. **Prevention** techniques are able to protect clients from being harmed by even unknown attacks.

### 2.4.2 Detection techniques

The detection of JavaScript-based threats requires the analysis on JavaScript code. Fig. 2.5 shows that all detection approaches can be generally classified into three classes, according to the analysis techniques that they use:

1. Static analysis

   Static analysis does not require code execution. Static analysis usually analyses the syntax of the code (e.g., lexical analysis, syntactic analysis, such as type-based analysis). The goal is to check whether suspicious keywords or code fragments exist. Also semantic analysis techniques have been proposed (e.g., points-to analysis helps understanding the points-to relationship between the heap objects). There are mainly 3 kinds of static analysis:

   (a) Type-based

   Object-oriented scripting languages like JavaScript is popular partly because of the dynamic features. These include the runtime modification of objects and classes through addition of fields or updating of methods. These features make static typing difficult. The idea is to statically define structural types to JavaScript. The types should allow JavaScript objects to evolve in a controlled manner. It is necessary to define a type inference algorithm for JavaScript that is sound with respect to the type system. If the type inference algorithm succeeds, then the program is type-able. Therefore, programmers can benefit from the safety offered by the type system, without the need to write explicitly types in their programs.

   [Jensen et al., 2009] introduces type analysis into JavaScript. Type analysis can be used for code correctness checking, and it can also be used for security reasons by high-levelling JavaScript code into security-related types such as described in [Politz et al.,

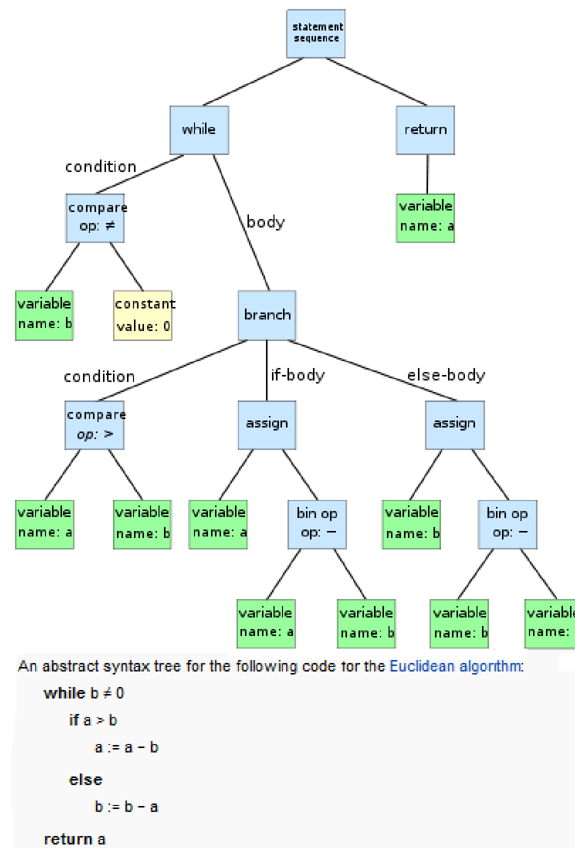


Figure 2.6: An example of AST. Source: `http://en.wikipedia.org/wiki/Abstract_syntax_tree`

2011]. [Anderson et al., 2005] introduces a type inference algorithm, which is the first paper that introduces a set of types defined for JavaScript.

(b) AST-based

Abstract Syntax Tree (AST) is a tree representation that presents the abstract syntactic structure of source code as exemplified in Fig. 2.6. All constructs such as variables, values, keywords are contained in the tree in the form of AST nodes. AST-based analysis is a very popular technique that is used to extract static features such as the usage of keywords or the suspicious variable assignment. For instance, ZOZZLE [Curtsinger et al., 2011] uses machine learning on features, which are extracted from AST nodes, to build model for benign JavaScript and Gatekeeper [Guarnieri and Livshits, 2009] passes JavaScript programs' AST for further points-to analysis and policy enforcement. Unfortunately, AST-based analysis can not help to understand what is happening in the heap, what is the JavaScript code exactly doing.

(c) Points-to analysis

As exemplified in Fig. 2.7, points-to analysis is a static code analysis technique that establishes, which pointers, or heap references, can point to which variables or storage locations. Since it provides the information of the heap and the object references, it has semantic meanings. [Jang and Choe, 2009] introduces how points-to analysis is ported from C to JavaScript. Paper [Barth et al., 2009] uses points-to analysis to monitor the JavaScript heap and draw heap graph, in which suspicious edges are searched for. Paper [Taly et al., 2011] proposes to use points-to analysis to monitor JavaScript API and implement API confinement.

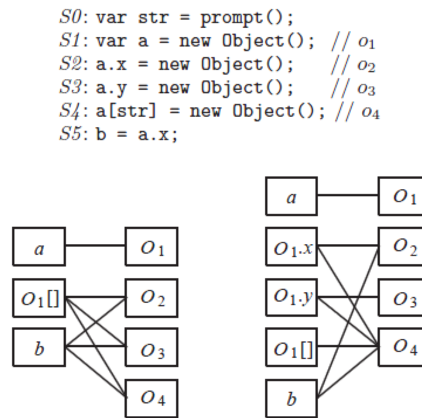


Figure 2.7: Example of JavaScript program and its points-to graphs. Top: program code, bottom left: conventional graph, bottom right: graph with properties. Source: [Jang and Choe, 2009]

**Conclusion** Type-based analysis is not widely used for security purpose (1 referenced paper) as it requires a good definition of types, which is very difficult. However, it can be used as a pre-processing filter because it requires less computing resources.

AST-based analysis is commonly used and quite well developed (3 referenced papers). It can ease the representation of JavaScript program for other analysis techniques or it can extract static features on which machine learning can build knowledge model.

Points-to analysis is also common (4 referenced papers). It is used especially when memory/heap information, such as variable assignment and string allocation, is crucial to be known. It can help researchers to know what the code is actually doing.

2. Dynamic analysis

   Compared to static analysis, the idea of dynamic analysis is very simple: executing the code, logging the interesting events and analysing the logs. Dynamic analysis requires partial or complete emulation of a real client (e.g., file system, browser environment, plug-ins). We classify all mitigation approaches that uses dynamic analysis into 3 subclasses, according to different emulation techniques they use:

   (a) VM-based

   The approaches that use VM-based emulation techniques are typically called honeypots, or honeyclients. Honeypots and honeyclients are usually divided into either low-interaction or high-interaction: High-interaction honeypots are real systems, providing real applications for the malicious code to interact with. Low-interaction honeypots emulate real systems and services. High-interaction honyclients are usually real automated web browsers on real operating systems, which allows and traces the execution of malicious code. Low-interaction honyclients, on the other hand, are usually emulated web browsers.

   Capture-HPC [Christian Seifert, 2006], which is widely used for research papers, is a famous example of honeypots.

   (b) Sandbox

   Sandbox, such as[Microsoft, 2012], is a security mechanism to isolate the execution of untrusted JavaScript in a virtualizd runtime browser environment. Usually, JavaScript sandboxes are implemented inside the browser to virtualize and restrict the access to

DOM (Document Object Model, which is a cross-platform and language-independent convention for representing and interacting with objects in HTML, XHTML and XML documents.) or JavaScript API (e.g., root object `Document`, or `Doucument.write()`). In practice, to decide whether a piece of JavaScript code is malicious or not, the restricted DOM or JavaScript API is monitored to check whether code behaviours trigger pre-defined heuristic. For example, ADSandbox [Dewald et al., 2010] propose a hybrid solution to detect malicious JavaScript, which uses sandbox as its dynamic analysis component. The pre-defined heuristics in ADSandbox include restricting the usage of `SaveToFile()`.

(c) Instrumented browser, JavaScript interpreter

Other approaches that use dynamic analysis rely on using instrumented browsers and JavaScript interpreters to log information (e.g., JavaScript variables, function invocation chain). For example, Caja [Mark S. Miller, 2007] propose a new secure subset of JavaScript to safely combine untrusted third party code (described in Subsection 2.3.1). Since Caja changes the specifications of JavaScript, to compile Caja subset, it requires the corresponding modifications on browser interpreter.

**Conclusion** Compared to other two kinds of dynamic analysis techniques, VM-based approaches are much more expensive, since it requires much more computing resources to virtualize a complete machine. Another limitation is that it is very difficult to cover all the possible configurations of real computers. For example, it is possible that JavaScript-based attacks, which target vulnerabilities of Firefox and Windows OS, can successfully evade honeypot/honeyclient detection if honeypot/honeyclient does not consider Firefox + Windows OS configuration. An advantage of VM-based dynamic analysis is that they can monitor the whole system: even a slight change on the file system or on the kernel level can be captured, which cannot be achieved by other kinds of approaches.

Sandbox, such as ADsandbox [Dewald et al., 2010], js.js sandbox[Terrace et al., 2012], is a good way to limit the privileges of executing JavaScript and protect the access to privacy-related information, but there is a limitation that scripts may conform to the sandbox policy, but still violate the security of the system. For example, scripts may abuse systems resources, such as opening browser windows that never close or creating a large number of pop-up windows.

Instrumented Browser: There are papers such as Paper [Hallaraker and Vigna, 2005], Paper [Dhawan and Ganapathy, 2009], Paper [Egele et al., 2009] use this technique. The advantage of this technique is that it reuse the existing browser interpreter, which means reducing effort to build or virtualize a new browser. But the limitation is that it can only monitor the limited information/objects/variables inside the browser.

The most important aspect for all dynamic analysis approaches is what is monitored and logged. Generally, there are 2 kinds of information that is considered interesting for security reasons and is logged: (1) Security-related functions (e.g., `eval()`, `document.write()`), JavaScript API and DOM (e.g., `Document`) (2) Changes of the file system or the state variables of the browser or the states of the OS kernel.

3. Hybrid analysis

   There are many ways to combine static and dynamic analysis techniques, such as:

| | Advantages | Limitations |
|---|---|---|
| Static analysis | Faster<br>Complete code convergence | Can not do de-obfuscation<br>Can not analyse dynamically generated code |
| Dynamic analysis | Can handle obfuscation<br>Can analyse dynamically generated code | Slower<br>Can not cover all the code |

Table 2.1: Comparison between static analysis and dynamic analysis.

(a) Use static analysis for filtering obvious malicious JavaScript then use dynamic analysis only for suspicious JavaScript as [Canali et al., 2011] does.

(b) Use dynamic analysis only for de-obfuscation as first step then apply static analysis (e.g., ZOZZLE[Curtsinger et al., 2011]).

(c) Features used for machine-learning, which require using both static analysis and dynamic analysis (e.g., Paper [Likarish et al., 2009], Wepawet [Cova et al., 2010], Cujo [Rieck et al., 2010]).

We describe in detail how static analysis and dynamic analysis are combined with example papers in Chapter 3

Another type of hybrid analysis is data-flow analysis. Data-flow analysis is a technique for gathering information about the possible set of values calculated at various points in a program. In JavaScript data-flow analysis is extended to Internet data flow (e.g., the invocation of a variable at source B from a piece of JavaScript code at source A). This is achieved in three steps: (1) Inserting data-taints: all variables or browser DOMs are tainted when modified or declared. (2) data-taints propagation: if a variable used in an expression that sets a second variable that second variable is now tainted as the first variable. (3) data-taints checking: according to pre-defined heuristics, taints are checked at the end. Data-flow analysis is especially useful for mitigating mash-ups or malicious JavaScript extensions (both are described in Subsection 2.3.1).

Data-flow analysis is a hybrid analysis, because inserting data-taints is achieved by using dynamic source code rewriting, also the data taints propagation needs the help of dynamic execution of tainted JavaScript, but after all, the checking method in most case is static analysis by checking the taints. There are quite a lot of papers that choose using this technique such as Paper [Vogt et al., 2007], Spectator [Livshits and Cui, 2008], Paper [Jang et al., 2010], Paper [Chugh et al., 2009], VEX [Bandhakavi et al., 2010], and [Djeric and Goel, 2010].

**Comparison between static analysis and dynamic analysis** Table 2.1 describes the advantages and limitations of static analysis and dynamic analysis. Static analysis can not mitigate the attacks targeting runtime code execution such as obfuscation, but it can cover all possible code paths. Dynamic analysis can mitigate the attacks such as obfuscation as a complementary solution to static analysis, but it can only analysis the code that is executed at run-time, which means partial of the code, which is not covered and executed can not be analysed by dynamic analysis.

### 2.4.3 Prevention techniques

Prevention techniques aim at protecting clients without analysing JavaScript code. Mainly there are 4 existing prevention techniques:

1. Code rewriting

   Code rewriting is a very general prevention technique. The goal is to rewrite the potentially-malicious code into safe code. Usually code rewriting technique is applied with defining new secure subset of JavaScript code together. Another usage of code rewriting is the enforcement of policies, which means inserting policy-check code into source code.

2. Defining secure subset of JavaScript

   The purpose of this prevention technique is to enforce the usage of a secure subset of JavaScript. One example is FBJS [Facebook, 2012], which is a subset of JavaScript defined by Facebook, developers are forced to use FBJS to write web applications for facebook. ADsafe, another subset of JavaScript that is powerful enough to allow third-party code to perform valuable interactions, while at the same time preventing malicious or accidental damage or intrusion.

3. Filtering

   In JavaScript security field, filtering is a common technique to avoid executing obviously malicious JavaScript code. The decision relies on pre-defined heuristics, which can be matching code-fragment patterns or keywords filtering (e.g., no use of `eval()`).

4. Wrapping

   Wrapping has two usages: JavaScript API wrapping and browser DOM wrapping. JavaScript API wrapping is used usually to protect the usage of security-related functions such as `eval()` by transform it into another new function, developer will be forced to use the new function instead of the original one, during the execution of the code, new function will be compiled back to the original function first, then interpreted by browser. The idea of browser DOM wrapping is similar to JavaScript API wrapping, original DOM has its own image during the code execution. All modifications act on the image DOM. This technique requires to instrument the browser interpreter.

In practice, these prevention techniques are usually used together. There are 7 papers: BrowserShield [Reis et al., 2006], Caja [Mark S. Miller, 2007], paper [Maffeis et al., 2009], paper [Phung et al., 2009], paper [Sergio Maffeis, 2009], paper [Dongseok Jang, 2010], paper[Finifter et al., 2010] that describe how these techniques work together. We will discuss these papers in detail in Chapter 3.

## 2.5 Correlation between threats and mitigation techniques

In this section, we describe what class of mitigation approaches are effective against what combination of different attacking **vectors**, **evasions** and goals. To this end, we organized all referenced papers into two tables (Table 2.2 and 2.3), from which we can observe that:

1. Data-flow analysis is widely used for detecting malicious code that leverage JavaScript extensions, mash-ups and inclusion of code from untrusted third parties.

2. Defining secure subset of JavaScript is useful to mitigate mash-ups or untrusted third party code.

3. Rewriting/Filtering/Wrapping is usually combined and used with data-flow and defining secure subset of JavaScript.

4. Pure static analysis is not widely used.

5. Hybrid and dynamic analysis are widely used, and machine learning are introduced in from 2009.

6. In mitigation approaches that use hybrid analysis, static analysis component usually extracts static features for machine learning or statically analyse the report that is generated by dynamic analysis component.

| | None | Obfuscation | Environment Matching |
|---|---|---|---|
| XSS | Dynamic analysis:<br>[Hallaraker and Vigna, 2005]<br>[Kirda et al., 2006]<br>[Cao et al., 2012]<br>Hybrid/Data-flow:<br>[Vogt et al., 2007] | | |
| Mash-up | Static analysis:<br>[Barth et al., 2009]<br>[Guarnieri and Livshits, 2009]<br>[Finifter et al., 2010]<br>[Taly et al., 2011]<br>Dynamic analysis:<br>[Terrace et al., 2012]<br>Hybrid/Data-flow:<br>[Chugh et al., 2009]<br>[Dongseok Jang, 2010]<br>Prevention::<br>[Phung et al., 2009]<br>[Maffeis et al., 2009]<br>[Maffeis et al., 2010]<br>[Mark S. Miller, 2007]<br>[Dongseok Jang, 2010]<br>[Finifter et al., 2010]<br>[Chugh et al., 2009] | | |
| JavaScript Extension | Hybrid/Data-flow:<br>[Dhawan and Ganapathy, 2009]<br>[Bandhakavi et al., 2010]<br>[Djeric and Goel, 2010] | | |
| Malicious JavaScript | Prevention:<br>[Reis et al., 2006]<br>Hybrid/Data-flow:<br>[Livshits and Cui, 2008]<br>[Jang et al., 2010]<br>[Curtsinger et al., 2011]<br>Dynamic analysis:<br>[Dewald et al., 2010]<br>[Reis et al., 2006] | Hybrid:<br>[Likarish et al., 2009] | Dynamic analysis:<br>[Kolbitsch et al., 2012] |
| PDF with embedded JavaScript | Static analysis:<br>[Laskov and Srndic, 2011]<br>Hybrid:<br>[Tzermias et al., 2011] | | |
| None | | Dynamic analysis:<br>[Peck, 2007]<br>Hybrid:<br>[Lu and Debray, 2012] | |

Table 2.2: Papers the propose mitigation techniques against **vectors** and **evasions**.

| | None | Obfuscation | Heap Spraying | Environment Matching |
|---|---|---|---|---|
| Drive-By Download | Hybrid:<br>[Rieck et al., 2010]<br>Dynamic:<br>[Song et al., 2010]<br>[Heiderich et al., 2011] | Hybrid:<br>[Cova et al., 2010] | Dynamic:<br>[Egele et al., 2009] | Hybrid:<br>[Cova et al., 2010] |
| None | | | Hybrid:<br>[Ratanaworabhan et al., 2009] | |

Table 2.3: Papers the propose mitigation techniques against goals, **techniques** and **evasions**.

# Chapter 3

# Detailed survey and comparison of research approaches

In this chapter, we describe in detail all referenced papers that propose mitigation approaches. We organize these papers according to the targets that their approaches mitigate, which are **vectors**, **evasions**, **techniques** or sub-goal: drive-by download. Some papers target a combination of **vectors** and **evasions**; we explain these papers under the section of main target they address.

Since some papers consider malicious JavaScript itself without mitigating any specific **vector** or **evasion**, we describe these papers in an individual section: malicious JavaScript. Moreover, there are papers that target a more general goal, drive-by download, we decide to explain these papers together in another single section.

Besides the sections we have mentioned, the rest of the sections are:

- XSS vulnerabilities
- Environment matching
- Obfuscation
- JavaScript extensions
- Mash-ups
- PDFs with embedded JavaScript
- Heap spraying

At last, totally we have 9 sections in this chapter.

## 3.1 Malicious JavaScript

In this section, we describe 5 papers in detail that target malicious JavaScript without mitigating any specific **vector** or **evasion**. We find that all these papers propose hybrid analysis technique.

- **ADSandbox: sandboxing JavaScript to fight malicious websites** [Dewald et al., 2010]

  Classification:

    Hybrid analysis.

Description:

Fig. 3.1 shows the architecture of ADSandbox. The core of ADSandbox is a controlled execution environment (sandbox) for JavaScript. It utilizes Mozilla JavaScript engine, SpiderMonkey, to execute JavaScript programs and log code behaviour during the execution. Afterwards, the system uses predefined heuristics on the resulting log to detect malicious behaviour. All this is implemented within a browser helper object (BHO) that interrupts any navigation process and initiates the analysis of the target URL by the sandbox.

At the lowest level, two types of static analysis of the source code are performed: (1) static IFrame/same origin analysis that detects every IFrame on a website and analyses properties such as width, height, position to decide whether this IFrame comes from a website with malicious JavaScript code. (2) static JavaScript analysis that detects the manipulation of an object's prototype or the use of the `eval` function. The next level involves dynamic analysis on the behaviours of the JavaScript programs that are embedded in the website. The dynamic analysis is performed in 3 phases: (1) JavaScript source code is extracted from the HTML code, then passed and wrapped into JavaScriptExecution Object, which is newly defined. (2) The JavaScript execution creates a new instance of SpiderMonkey and executes the given "JavaScriptExecution Object". SpiderMonkey on the other hand, is instrumented to interpreter "JavaScriptExecution Object". (3) During execution, each time an JavaScript object is accessed, it will trigger a corresponding static callback function of the JavaScriptExecution Object, so every access to every JavaScript object is recognized and logged.

The execution log is searched for patterns that reveal typical malicious behaviour. These patterns are implemented as regular expressions, which allow efficient matching. In total, they define seven malicious behaviour patterns to detect a range of attacks such as cookie stealing, file downloads and heap-spraying attacks. For example, they search in the execution log if the JavaScript uses the function `SaveToFile()` or `run()`, which are often used together, to save byte-code into a file and then run this file. The corresponding regular expression pattern is `CONVERT (SaveToFile|Run) TO A FUNCTION`

- **Spectator: detection and containment of JavaScript worms** [Livshits and Cui, 2008]

Classification:

Hybrid analysis, Data-flow analysis.

Description:

Spectator is an automatic detection and containment solution for malicious JavaScript. Spectator performs data tainting (described here in Subsection 2.4.2) by observing and tagging the data-flow between the clients and the web application. When a data-taint propagates too far, a warning is reported.

The solution proposed in this paper seems quite similar to Paper [Vogt et al., 2007]. They both use data-flow analysis. The key difference is: in Paper [Vogt et al., 2007], data tainting is implemented by analysing JavaScript code on the client side. In this paper, it requires the cooperation both on the client side and on the server side. Spectator tags both HTTP request and response, the criteria is to find long propagation chain.

The scenario is shown is Fig. 3.2. Whenever a user attempts to download a page containing Spectator tags, the following steps are taken:

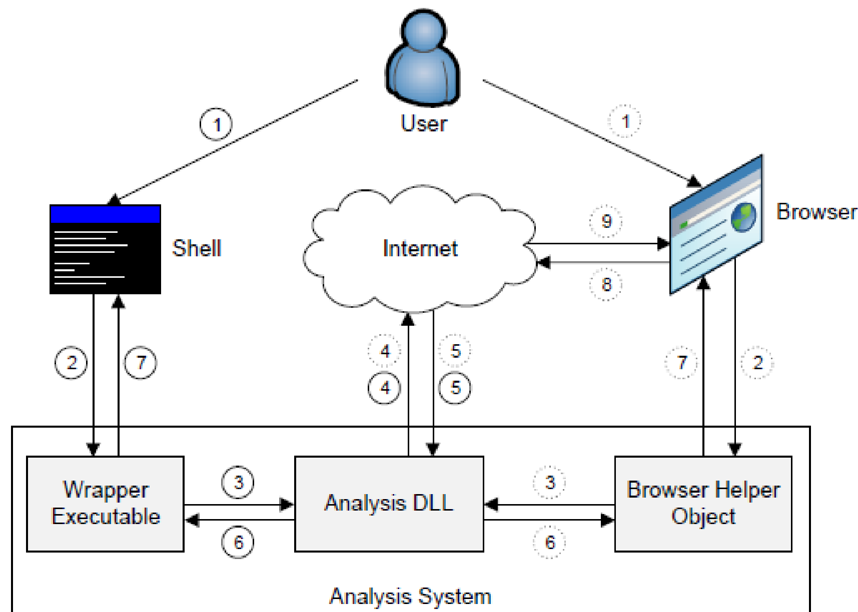


Figure 3.1: System overview of ADSandbox. Source: [Dewald et al., 2010].

1. The tagged page is retrieved from the server.
2. The Spectator proxy examines the page. If the page contains tags, a new session ID is created and associated with the list of tags in the page. The tags are stripped from the page and are never seen by the browser or any malicious content executing therein.
3. The modified page augmented with the session ID stored in a cookie (referred to below as Spectator cookie) is passed to the browser.

Whenever an upload containing HTML is observed, the following steps are taken:

1. The client issues an HTTP request containing HTML and a new tagged content (e.g., a new piece of JavaScript code that is tagged) for that upload. If a Spectator cookie is found on the client, it is automatically sent to Spectator by the browser.
2. if the request has a valid session ID contained in a Spectator cookie attached to the request, the list of tags it corresponds is looked up and, for every tag, links are added to the propagation graph. The request is not propagated further if the Spectator detection algorithm decides that the request is part of malicious code propagation.
3. Finally, the request augmented with the newly created tag is uploaded and stored at the server.

The advantages of Spectator are:

1. Spectator, not only can detect malicious JavaScript, but also contain the propagation of malicious JavaScript code.
2. Spectator can deal with rapid zero-day attacks as well as malicious code that disguise their presence with slow propagation.

- **An empirical study of privacy-violating information flows in JavaScript web applications** [Jang et al., 2010]

  Classification:

  Hybrid analysis, data-flow analysis.

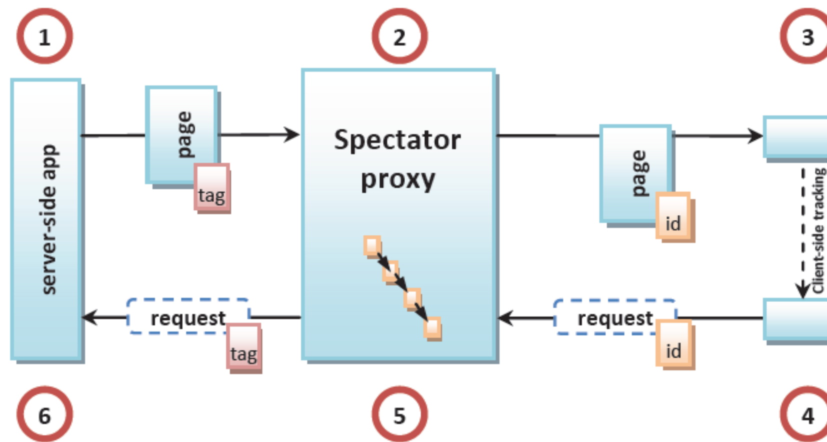


Figure 3.2: Spectator architecture. Source: [Livshits and Cui, 2008].

Description:

This approach uses data-flow analysis and it consists of three steps:

1. designed an expressive, fine-grained information flow policy language that allows to specify and detect different kinds of privacy-violating flows in JavaScript code. A fine-grained information flow policy is specified by defining taints, injection taints and checking taints. A taint can be any JavaScript object, e.g., a URL string denoting the provenance of a given piece of information. Policy is enforced by automatically rewriting the code using the specified injection and checking taints.
2. implement a new rewriting-based JavaScript information flow engine within the Chrome browser.
3. used the enhanced browser to conduct a large-scale empirical study over the 50,000 websites of four privacy-violating flows: cookie stealing, location hijacking, history sniffing, and behaviour tracking.

There are 4 important policies that are considered:

1. `document:cookie` should remain confidential.
2. `document:location` should not be influenced.
3. same origin/domain policy (same as described in paper[Hallaraker and Vigna, 2005].
4. white-list policy (If a JavaScript's origin is within the white-list predefined, the JavaScript is considered safe anyway).

- **ZOZZLE: fast and precise in-browser JavaScript malware detection** [Curtsinger et al., 2011]

Classification:

Hybrid analysis.

Description:

ZOZZLE is a mostly static JavaScript malware detector that is fast enough to be used in a browser. While its analysis is entirely static, ZOZZLE has a runtime component to address the issue of JavaScript obfuscation (described here in Subsection 2.3.1). The dynamic analysis component monitors functions such as `eval()` and `document.write()`.

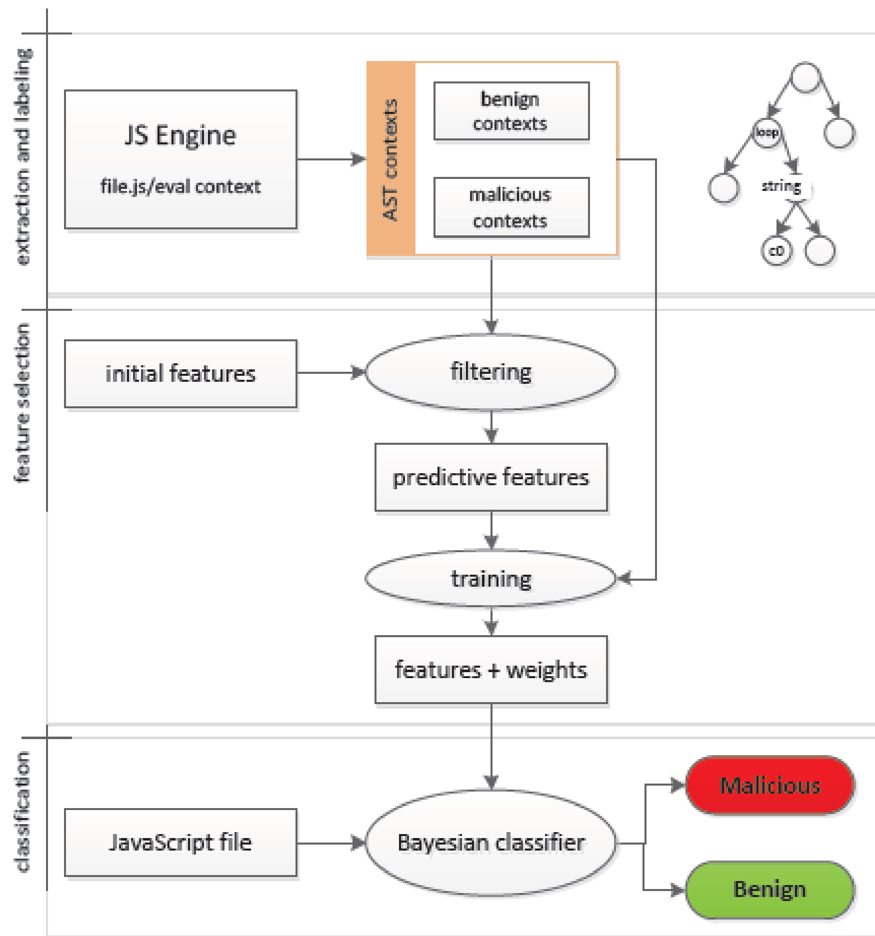


Figure 3.3: ZOZZLE training process. Source: [Curtsinger et al., 2011].

The unfolded JavaScript then is passed to a static classifier that is trained based on features that are extracted from JavaScript AST (described herein Subsection 2.4.2)

Machine learning is applied to learn the characteristic features of malicious JavaScript. Specifically, a feature consists of two parts: a context in which it appears (such as a loop, conditional, try/catch block, etc.) and the text (or some sub-string) of the AST node. To limit the possible number of features, only features from specific nodes of JavaScript AST are extracted: expressions and variable declarations. Finally, they use a Bayesian classifier to build the knowledge model. Fig. 3.3 illustrates the learning process.

The accuracy of ZOZZLE relies on the correctness of the knowledge learned by machine learning from malicious JavaScript samples. On another word, it highly depends on the feature robustness of the knowledge model.

- **BrowserShield: vulnerability-driven filtering of dynamic HTML** [Reis et al., 2006]

  Classification:

  Prevention, dynamic analysis (sandbox)

  Description:

  The approach proposed in this paper is to translate HTML pages and any embedded scripts including JavaScript into safe equivalents before they are rendered by the browser. The safe equivalent pages contain logic to recursively apply runtime checks

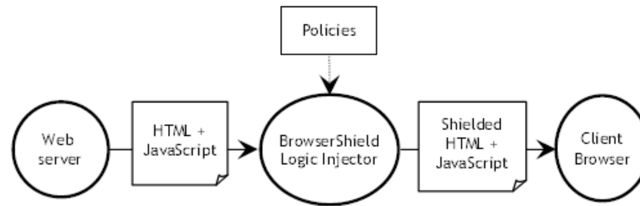


Figure 3.4: Deployment of BrowserShield. Source: [Reis et al., 2006].

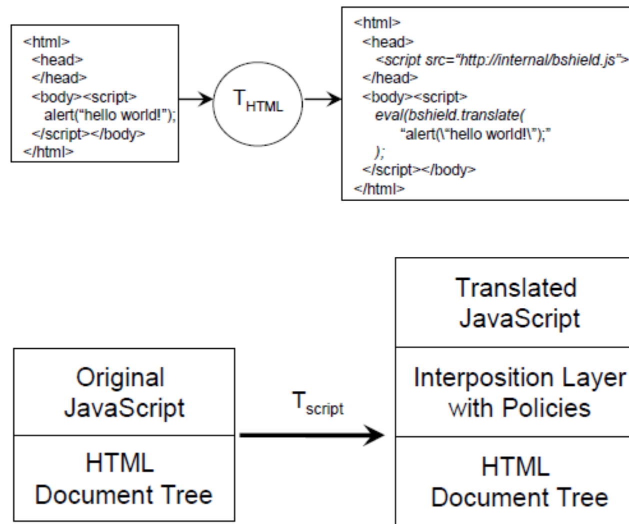


Figure 3.5: HTML and script translation of BrowserShield. Source: [Reis et al., 2006].

to dynamically generated or modified web content, based on known vulnerabilities (which is also a limitation for BrowserShield). Fig. 3.4 shows the deployment of BrowserShield to protect the client.

In BrowserShield, the translation is done separately in two processes. One is HTML translation, another is script translation, we use Fig. 3.5 to illustrate examples for translation.

To achieve complete interposition, runtime checks are injected to interpose on function calls, JavaScript object method calls, JavaScript object property accesses, JavaScript object creation, and control constructs.

## 3.2 XSS vulnerabilities

In this section, there are 3 papers regarding mitigation against XSS vulnerabilities from 2005 to 2012. All these 3 approaches need to use dynamic analysis technique.

- **Detecting Malicious JavaScript Code in Mozilla** [Hallaraker and Vigna, 2005]

  Classification:

  Dynamic analysis.

  Description:

  This paper is the first paper that gazes on JavaScript security issues. It rises the problem of malicious JavaScript injection through XSS vulnerabilities, and the solution

that it proposes uses dynamic analysis (instrumented interpreter) based on monitoring JavaScript code execution and compare the execution behaviour to pre-defined high-level policies. In detail, this approach can be divided into 2 components: auditing system and IDS (Intrusion Detection System). The auditing system is responsible for collecting information about JavaScript execution. Based on these information, IDS can judge whether the running JavaScript code is malicious or not. The auditing system is integrated with Spider Monkey, a JavaScript Interpreter, to audit method calls and property getters and setters. For the IDS, they propose a policy-based IDS, and the two most important policies are same-origin policy and signed-script policy. The same-origin policy prevents documents or scripts loaded from one origin (i.e., a web server), from getting or setting properties of a document from a different origin. In this context, same origin means same protocol, host, and port. This policy provides the foundation for isolating one script from another, and ensures that a document downloaded from one source cannot be changed by JavaScript code downloaded from another origin. The signed-script policy was developed to give JavaScript more functionality and give users the option to define a finer-grained security policy. Script signing allows a script to get out of the auditing and is similar to the mechanisms used for signed Java applets.

The key difference between this approach and sand-boxing mechanism such as AD-Sandbox[Dewald et al., 2010], paper [Terrace et al., 2012] is that sandbox restricts the access to the resources such as API and private data, but the authority given by sandbox is not based on the dynamic behaviour of JavaScript code, so the authority might be given in a mistake that cause a successful attack (an example is described here in Subsection 2.4.2). But this approach suggests logging all the actual behaviour of the JavaScript code, so it will avoid this problem if necessary behaviour is logged and the auditing system is perfect for all the attacks (which in reality the auditing system is not capable to detect all the attacks).

The main limitation of this approach is that it is only applicable to protect known vulnerabilities and the performance highly depends on the efficiency of the auditing system.

- **Pathcutter: Severing the self-propagation path of XSS JavaScript worms in social web networks** [Cao et al., 2012]

  Classification:

    Dynamic analysis.

  Description:

    The tool that this paper propose uses dynamic analysis, which aims at blocking the propagation of malicious JavaScript through XSS vulnerabilities. Pathcutter works by blocking two critical steps in the propagation path: (i) DOM access to different views (a portion of a web application) at the client side and (ii) unauthorized HTTP request to the server. To achieve these two goals, Pathcutter proposes two integral mechanisms: view separation and request authentication. For view separation, they use a solution proposed by MiMoSA [Balzarotti et al., 2007] and for request authentication this paper makes use of [Barth et al., 2008].

    The working progress of Pathcutter is:

    1. Dividing a web application/webpage into different views.
    2. Isolating the different views at the client side.

| Object | Tainted properties |
|---|---|
| Document | Cookie, domain, forms, lastModified, links, referrer, title, URL |
| Form | action |
| Any form input element | checked, defaultchecked, defaultvaule, name, selectedindex, tostring, value |
| History | current, next, previous, tostring |
| Select option | defaultselected, selected, text, value |
| Location and link | hash, host, hostname, href, pathname, port, protocol, search, tostring |
| Window | default status, status |

Table 3.1: Initial source of tainted values. Source: [Vogt et al., 2007].

3. If the request is from a view that has no right to perform a specific action, the request is denied, which means the potential propagation is stopped.

This paper is interesting because it is the only paper we found that targets stopping the propagation of malicious JavaScript.

- **Cross-site scripting prevention with dynamic data tainting and static analysis** [Vogt et al., 2007]

  Classification:

  Hybrid analysis, Data-flow analysis.

  Description:

  The solution proposed in the paper uses data-flow analysis. The information flow of sensitive data is tracked inside the JavaScript engine of the browser. Unfortunately, it is not possible to detect all information flows dynamically. To address this limitation, an additional static analysis component is used to complement the dynamic mechanism. This static analysis component is invoked on-demand and covers those cases that cannot be decided dynamically. The part of dynamic analysis allows to precisely track sensitive information with low runtime overhead. By switching to static analysis when necessary, the system can provide stronger security in the face of malevolent attack code.

  Table. 3.1 shows all the sensitive information that is tracked. Also, JavaScript programs that are part of a web page are parsed and compiled into an internal byte code representation. These byte code instructions are then interpreted by the JavaScript engine. To track the use of sensitive information by JavaScript programs, they have extended the JavaScript engine. More precisely, semantics of the byte code instructions has been extended so that taint information is correctly propagated.

  When tainted data (described here in Subsection 2.4.2) is about to be transferred to third party code (described here in Subsection 2.3.1), different kinds of actions can be taken. Examples are logging, preventing the transfer, or stopping the program with an error.

## 3.3 Environment matching

There is a very important paper that target this **vector**: environment matching. It is proposed in 2012. To detect the behaviour of environment matching, it does require code execution.

- **Rozzle: De-cloaking Internet Malware** [Kolbitsch et al., 2012]

Classification:

Dynamic analysis (VM-based).

Description:

The security issue that this paper focuses on is environment matching /environment-specific attack. To solve this issue, this paper proposes Rozzle, a JavaScript multi-execution virtual machine (VM-based), as a way to explore multi-path execution. Rozzle is an enhancement or amplification technology, designed to improve the efficiency of both static and runtime JavaScript malware detection.

The key technique that is used by Rozzle is **Symbolic Execution**. **Symbolic Execution** [Saxena et al., 2010]refers to the analysis of programs by tracking symbolic rather than actual values, a case of abstract interpretation. The key idea behind Rozzle is to execute all possible code paths whenever it encounters control flow branching that is dependent on the environment. For example, in the case of an `if` statement, Rozzle will execute both branches, one after another. During code multi-path execution, weak updates are performed, in other words, the second assignment to variable does not override, but adds to the first value.

A limitation of Rozzle is that it can be evaded by server-side cloaking (described here in Subsection 2.3.1), or if Rozzle itself is detected.

## 3.4 Obfuscation

Obfuscation is a commonly used technique. In this section, we describe 3 papers that target it. As we have explained, obfuscation is an example to show the dynamic feature of JavaScript, so to detect obfuscated malicious JavaScript code, the mitigation approach must at least contain one dynamic analysis component.

- **Caffeine Monkey** [Peck, 2007]

  Classification:

  Dynamic analysis (sandbox).

  Description:

  This paper introduces the obfuscation techniques that they address: white-space randomization, block randomization and the usage of some common functions for obfuscation such as `eval(),document.write(),String.fromCharcode()`. In this paper a sandbox is used to execute JavaScript code and during the execution the final unfolded JavaScript code can be recorded. To detect malicious JavaScript, the function calls are counted. If the percentage of specific function calls is above a threshold, then this piece of JavaScript code is supposed to be malicious.

  This paper also does a survey on the wild using their approach, they found that benign JavaScript makes significant use of `document.write()` method while malicious JavaScript makes relatively more use of string instantiation.

  This paper is the first paper that formally address the problem of obfuscation and provides a simple, basic solution for further improvement.

- **Obfuscated malicious JavaScript detection using classification techniques** [Likarish et al., 2009]

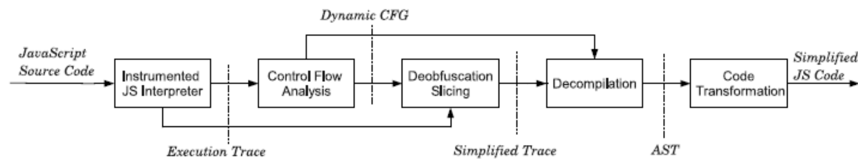


Figure 3.6: De-obfuscation process. Source: [Lu and Debray, 2012].

Classification:

Hybrid analysis.

Description:

The approach proposed by this paper is very similar to Wepawet [Cova et al., 2010]. It is also based on applying machine learning to the features of obfuscated malicious JavaScript. There are two differences that varies from this paper to Wepawet [Cova et al., 2010]: First, they extract 65 features in this paper, which are different (JavaScript keywords and symbols accounted for 50 of them). Among all the features, there are 5 features that are most highly correlated with malicious JavaScript:

1. human readable
2. the use of the JavaScript keyword `eval`
3. the percentage of the script that was white-space
4. the average string length
5. the average characters per line

The features they extract require the usage of both static analysis and dynamic analysis. Second, they use not only one machine learning technique, but Naive Bayes, Alternating Decision Tree (ADTree), Support Vector Machines (SVM) together and the RIPPER rule learner is used as a classifier module.

- **Automatic Simplification of Obfuscated JavaScript Code: A Semantics-Based Approach** [Lu and Debray, 2012]

Classification:

Hybrid analysis.

Description:

This paper propose another approach to de-obfuscate JavaScript code. Fig. 3.6 shows the de-obfuscation process, which consists of the following steps:

1. Use an instrumented interpreter to obtain an execution trace (including function calls, global variables, function references, `document.write()`, document elements and unfolded code) for the JavaScript code under consideration.
2. Construct a control flow graph from this trace to determine the structure of the code that is executed.
3. Use dynamic slicing algorithm to identify instructions that are relevant to the observable behaviour of the program. Ideally, computing slices for the arguments of the system calls made by the program. However, the actual system calls are typically made from external library routines that appear as native methods. As a proxy for system calls, therefore, the implementation of this paper computes slices for the arguments passed to any native function.

4. Decompile execution trace to an AST, and label all the nodes constructed from resulting set of relevant instructions.
5. Eliminate `goto` statements from the AST, then traverse it to generate de-obfuscated source code by printing only labelled syntax tree nodes.

The advantage of this approach is that the code obtained is semantically equivalent to the original code but has obfuscations simplified away, thereby exposing the core logic of the computation performed by the original code. The resulting code can then be examined manually or fed to other analysis tools for further processing.

## 3.5 JavaScript extensions

For commercial reasons and to improve users experience, there are a lot of browser extensions that are developed by using JavaScript. We think this is one of the most discovered attacking **vectors** in real life. In this section, we prepare 3 papers. We find that data-flow analysis is effective against this **vector**, because it can monitor the data flow between browser extensions and browser interpreter to protect private data leaking by cutting the flow.

- **Securing script-based extensibility in web browsers** [Djeric and Goel, 2010]

  Classification:

  Hybrid analysis, Data-flow analysis.

  Description:

  The solution of this paper uses data-flow analysis. The approach divides code into 2 classes: privileged code and unprivileged code. It guarantees that tainted data (described here in Subsection 2.4.2) will not be executed as privileged code. Tainting all data from untrusted origins and propagating the tainted data throughout the browser provides a much stronger basis for making security decisions.

  The approach uses different policies based on the privilege level of the executing script. Unprivileged code is completely untrusted and may be malicious, so it requires unconditionally tainting all script variables created or modified by executing scripts originating from untrusted (tainted) documents. For privileged scripts, they use standard taint propagation rules (described here in Subsection 2.4.2) that mark the output of JavaScript instructions as tainted when the instruction inputs are tainted. Tainting allows to mark and track the influence of untrusted code throughout the browser.

  They use both compilation detector (a proactive measure to prevent tainted data from being compiled to privileged byte code, even if it is never executed) and invocation detector (monitor script execution for situations where tainted references to script or native functions are invoked inside the interpreter and result in the creation of privileged stack frames) as detection components.

- **Analysing Information Flow in JavaScript-Based Browser Extensions** [Dhawan and Ganapathy, 2009]

  Classification:

  Data-flow analysis.

  Description:

| Entity | Sensitive attributes/Method of access |
|---|---|
| 1. Document | cookie, domain, forms, lastModified, links, referer, title, URL |
| 2. Form | action |
| 3. Form input | checked, defaultChecked, defaultVaule,name, selectedIndex, value, value |
| 4. History | current, next, previous, toString |
| 5. Select option | defaultSelected, selected, text, value |
| 6. Location/Link | hash, host, hostname, href, pathname, port, protocol, search, toString |
| 7. Window | defaultStatus, status |
| 8. Files/Streams | nsIInputStream, nsIFileInputStream, nsILocalFile, nsIFile |
| 9. Passwords | nsIPasswordManager, nsIPasswordManagerInternal |
| 10. Cookies | nsICookieService, nsICookieManager |
| 11. Preferences | nsIPrefService, nsIPrefBranch |
| 12. Bookmarks | nsIRDFDataSource |

Table 3.2: Sensitive sources of Sabre. Source: [Dhawan and Ganapathy, 2009].

| Entity | Method of access |
|---|---|
| 1. Files/Process | nsIOutputStream, nsIFileOutputStream, nsIFile, nsIProcess, nsIDownload |
| 2. Network | nsIXMLHTTPRequest, nsIHTTPChannel, nsITransport |
| 3. DOM | Submission of sensitive DOM node over the network |

Table 3.3: Insensitive sinks of Sabre. Source: [Dhawan and Ganapathy, 2009].

The tool proposed in this paper is Sabre, which provides a data-flow analysis solution. It tracks data flow at the level of JavaScript instructions and does so within the browser. Sabre achieves three goals:

1. Monitor JavaScript execution. Sabre monitors all JavaScript code executed by the browser. This includes code in web applications, JavaScript extensions, as well as JavaScript code executed by the browser interpreter.
2. Ease action attribution. When Sabre reports an information flow violation by a JavaScript extension, an analyst may need to determine whether the violation is because of an attack or whether the offending flow is part of the advertised behaviour of the JavaScript extension. In the latter case, the analyst must white-list the flow. To do so, it is important to allow for easy action attribution, i.e, an analyst must be able to quickly locate the JavaScript code that caused the information flow violation and determine whether the offending flow must be white-listed.
3. Track data flow across browser subsystems (e.g., HTML, XUL and SVG elements and XPCOM).

To implement the data flow system, they modifies Spider Monkey, a JavaScript interpreter, to include security label/taint(described here in Subsection 2.4.2) for document object. They also modify the implementation of Spider Monkey to allow the propagation of security label/taint. Sabre detects flows from sensitive sources to low sensitivity sinks. Table 3.2 and 3.3 shows the considered sensitive sources and sinks. Sabre alerts for an data-flow violation only when an document object is modified by JavaScript extension.

- **VEX: vetting browser extensions for security vulnerabilities** [Bandhakavi et al., 2010]

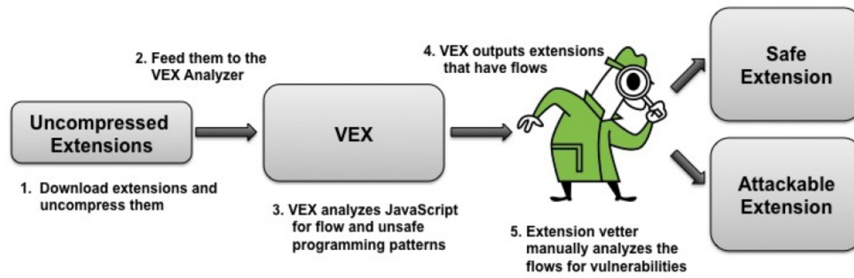


Figure 3.7: The overall analysis process of VEX. Source: [Bandhakavi et al., 2010].

Classification:

Data-flow analysis.

Description:

This paper proposes a solution similar to paper[Dhawan and Ganapathy, 2009]. They both check the data flow between the sources and the sinks that are important for security issues. The only difference is that the analysis is applied on a core subset of JavaScript (standard procedure of data-flow analysis is described here in Subsection 2.4.2) in this paper. The overall analysis process is shown in Fig. 3.7. Based on the result this analysis, five malicious patterns are defined for detection.

Specifically, VEX tracks flows from Resource Description Framework (RDF) data (e.g., bookmarks) to `innerHTML`, content document data to `eval()`, content document data to `innerHTML`, `evalInSandbox` return objects used improperly, or `wrappedJSObject` return object used improperly. For example, the source location is any point where the program accesses `window.content.document`, and the source object is the object that is returned from this call. The sink locations are `eval` statements and the sink objects are the objects being eval-ed.

RDF is a family of World Wide Web Consortium (W3C) specifications, originally designed as a metadata data model. It has come to be used as a general method for conceptual description or modelling of information that is implemented in web resources, using a variety of syntax notations and data serialization formats.

## 3.6 Mash-ups

As explained in Subsection 2.3.1. Mash-up is a **vector** that can cause object leakage to untrusted third party code. We find that this **vector** is also widely discovered in real life and there are lots of papers that target mitigating mash-ups. In this section, we select 11 papers. Mainly, there are 2 classes of solutions for mitigating this **vector**: (1) Define new secure subset of JavaScript, force developer to use new subset or rewrite original JavaScript code to new subset before rendering the code. (2) Data-flow analysis is used to monitor whether private data is leaked to untrusted third party.

- **JavaScript in JavaScript (js.js): sandboxing third-party scripts** [Terrace et al., 2012]

  Classification:

  Dynamic analysis (sandbox).

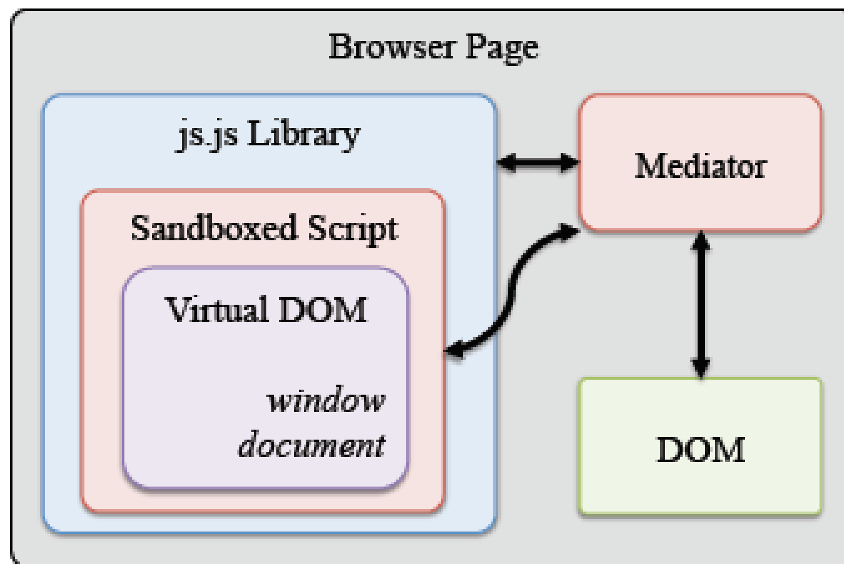


Figure 3.8: js.js architecture for example application. Source: [Terrace et al., 2012].

Description:

This paper proposes a sandbox. Fig. 3.8 shows an example of JavaScript application using js.js. A sand-boxed script has no access to any global variables except for JavaScript built-in types (e.g., Array, Date, and String), the JavaScript application can add additional names including global names like `window` and `document`. The js.js API allows an application, for example, to add a global name called `alert` that, when called inside the sandbox, calls a native JavaScript function. This way, the JavaScript application (the mediator in Fig. 3.8)using js.js has complete control over the sand-boxed script since the only access the sandbox is through these user defined methods. Thus these methods must give the script access only to the elements that the user allows.

- **Cross-origin JavaScript capability leaks: detection, exploitation, and defence** [Barth et al., 2009]

Classification:

Static analysis (points-to analysis).

Description:

The core idea of this paper is to get the points-to relationship (described here in Subsection 2.4.2) of JavaScript objects in the heap by using points-to analysis. From this relationship, they define the security origin of each JavaScript object by tracing its prototype chain. They then search for edges that connect objects in one security origin with objects in another security origin. These suspicious edges likely represent cross-origin JavaScript capability leaks.

They compute the security origin of each object directly from the is-prototype-of relation (as described here in Subsection 2.4.2) in the heap graph using the following algorithm:

1. Let `obj` be the JavaScript object in question.

2. If `obj` was created with a non-null prototype, assign `obj` the same origin as its prototype.

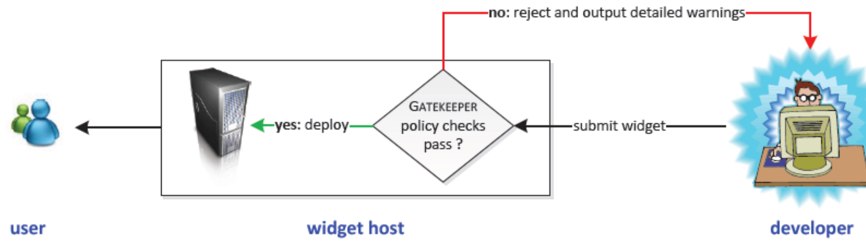


Figure 3.9: GATEKEEPER deployment. Source: [Guarnieri and Livshits, 2009].

3. Otherwise, `obj` must be the object prototype for some document `d`. In that case, assign `obj` the security origin of d (i.e., the scheme, host, and port of that `d's` URL).

Points-to analysis can generate heap graph, if you are interested to how to do points-to analysis, you can refer to [Jang and Choe, 2009].

Besides proposing a detection approach, this paper recommends that to mitigate capability leakage, access control checks should be added into JavaScript interpreter.

- **GATEKEEPER: mostly static enforcement of security and reliability policies for JavaScript code** [Guarnieri and Livshits, 2009]

  Classification:

  Static analysis (points-to analysis and AST-based analysis).

  Description:

  This paper proposes GATEKEEPER, a mostly static approach for enforcing security and reliability policies for JavaScript programs. Fig. 3.10 shows the analysis process of GATEKEEPER. The policies includes restricting widget capabilities, making sure built-in objects are not modified, preventing code injection attempts, redirect and cross-site scripting detection, preventing global name-space pollution, taint checking. Since JavaScript relies on heap-based allocation for the objects it creates, this paper uses points-to analysis on JavaScript. Since a sound and precise points-to analysis of the full JavaScript language is very hard to construct. Therefore, they propose a points-to analysis for JavaScript SAFE, a realistic subset that includes prototypes and reflective language constructs. To handle programs outside of the JavaScript SAFE subset, GATEKEEPER inserts runtime checks to preclude dynamic code introduction. On the basis of points-to information, they demonstrate the utility of their approach by describing nine representative security and reliability policies that are checked by GATEKEEPER, meaning no false negatives are introduced. For example, `alert` routine shall never be called and disallow changing properties of built-in objects such as `Boolean`, `Array`, `Date`, `Function`, `Math`, `Document` and `Window`. These policies are expressed in the form of succinct declarative data-log queries.

  The deployment of Gatekeeper is shown in Fig. 3.9, Gatekeeper is used as a verifier for widget before it is deployed on web server.

- **Automated Analysis of Security-Critical JavaScript APIs** [Taly et al., 2011]

  Classification:

  Static analysis(points-to based).

  Description:

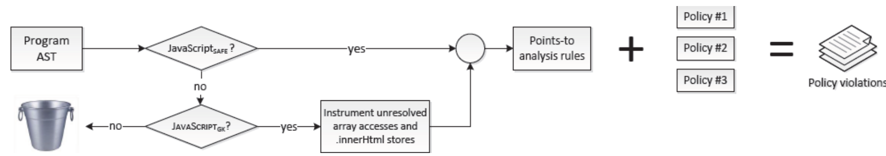


Figure 3.10: GATEKEEPER analysis architecture. Source: [Guarnieri and Livshits, 2009].

The purpose of this paper is to prevent privacy leakage to untrusted third party JavaScript code (described here in Subsection 2.3.1). The solution it proposes is to do points-to analysis for the purpose of JavaScript API confinement. In this paper, points-to analysis is applied on a subset of JavaScript ”SESlight”. SESlight solves two challenges related to the API Confinement: (1) All code has undeniable write access to the built-in objects, which can be maliciously used to alter the behaviour of trusted code that make use of built-in objects, and (2) code running inside `eval()` is unavailable statically, so it is hard to know what global state that is accessed. The first problem is solved by making all built-in objects, except the global object, transitively immutable, which means that all their properties are immutable and the objects cannot be extended with additional properties. Further, all built-in properties of the global object are made immutable. The second problem is addressed by imposing the restriction that all calls to `eval()` must specify an upper bound on the set of free variables of the code being eval-ed.

After doing points-to analysis on a piece of JavaScript code, a conservative data-log model of all API methods is generated. Besides, they encode attacker’s behaviours as a set of data-log rules, whose consequence set is an abstraction of the set of all possible invocations of all the API methods.

- **Staged information flow for JavaScript** [Chugh et al., 2009]

  Classification:

  Data-flow analysis.

  Description:

  This paper presents a data-flow analysis based approach for inferring the effects that a piece of JavaScript has on the website, in order to ensure the confidentiality of private data. Data flow can capture the fact that a particular value in the program affects another value in the program. To handle dynamically loaded and generated JavaScript code, they propose a framework for staging data flow properties. Staging consists of statically computing as much of the data flow as possible based on the known code, and leaving the remainder of the computation until more code becomes available. The framework propagates data flow through the currently known code in order to compute a minimal set of syntactic residual checks that are performed on the remaining code when it is dynamically loaded. If a piece of dynamically generated code is filled with other dynamically loaded code, then the staging framework recursively checks the residual policy.

  Their approach involves 2 related areas of research. First, they define a subset of JavaScript, they call it ”Core JavaScript,” which capture the essence of JavaScript. Second, in this approach, it provides a flow policy (refers to the residual policy in last paragraph), which is a set of pairs of policy elements. A policy element is a program variable or a hole (dynamic, unknown code). Each pair in the policy represents flow

that is disallowed. The idea of pairs and the expression of JavaScript code comes from set-constraint system, which you can refer to [Flanagan and Felleisen, 1999] and [Kodumal and Aiken, 2005].

- **Caja: Safe active content in sanitized JavaScript** [Mark S. Miller, 2007]

  Classification:

  Prevention.

  Description:

  This paper propose a new subset of JavaScript: Caja to allow third party JavaScript code safely combined into webpage. The main differences between Caja and complete JavaScript are:

  1. Forbidden names. Caja rejects all names ending with "__ " (double underscore).
  2. Frozen objects. If an object is frozen, an attempt to set, add, or delete its properties will throw an exception instead. Functions and prototypes are implicitly frozen. In addition, the Caja programmer can explicitly freeze objects to prevent their direct modification. All objects in the default global environment are immutable, or transitively frozen.
  3. No shared global environment. Each separately loaded code has its own global environment, which inherits from the default global environment, isolating them from each other.
  4. Internal names. Property names ending in "_" (single underscore) serve as protected instance variables. Such names can only appear to the right of "this.".
  5. Sharp knives removed. Caja contains no "`with`" or "`eval`".

  Although, Caja imposes 5 restrictions already. There are additional issues peculiar to JavaScript that must be dealt with, such as unconstrained properties of JavaScript objects. To solve these issues, they propose a subset of Caja: Cajita with more static and dynamic restrictions. For example, in Caja, an internal name is a property name ending in "_". Such names are used for encapsulation in Caja but are prohibited in Cajita. Cajita's only encapsulation mechanism is lexical scoping.

- **Isolating JavaScript with filters, rewriting, and wrappers** [Maffeis et al., 2009]

  Classification:

  Prevention.

  Description:

  This paper describes the guiding principles for all other papers related to prevention techniques. It formalize the problem of isolating JavaScript, and provide 3 techniques: filtering, rewriting, wrapping to achieve that goal, even more, it provides feasible rules to do filtering, rewriting and wrapping.

  For example, to isolate blacklisted (predefined) JavaScript object properties, it provides 3 filtering rules:

  1. Disallow all terms, which contain an identifier from the blacklist.
  2. Disallow all terms containing any of the identifier `eval`, `Function`, or `contructor`.
  3. Disallow all terms, which involve an identifier name beginning with $.

- **Lightweight self-protecting JavaScript** [Phung et al., 2009]

Classification:

Prevention.

Description:

This paper introduces a solution to control JavaScript execution. The aim is to prevent or modify inappropriate behaviour caused by e.g., malicious injected scripts or poorly designed third-party code. The approach is based on rewriting the code so as to make it self-protecting: the protection mechanism (security policy) is embedded into the code itself and intercepts security relevant API calls. The solution is lightweight in that (i) it does not require a modified browser, and (ii) it does not require any runtime parsing and transformation of code (including dynamically generated code). As a result, the method has low runtime overhead.

The security policies are based on the security relevant built-in method and the security states. JavaScript variables might be used to store security states for security decisions in security policies. For example, a policy: "application should never raise more than two pop-up windows" monitors the built-in method `window.open` and needs a security state variable to count the number of pop-ups so far. When all policy checks are embedded into the source code of a webpage, it is called self-protecting.

- **Run-Time Enforcement of Secure JavaScript Subsets** [Sergio Maffeis, 2009]

  Classification:

  Prevention.

  Description:

  This paper illustrates 2 fundamental issues of JavaScript isolation:

  1. Regardless of the techniques adopted to enforce isolation, the ultimate goal is: make sure that a piece of untrusted code does not access a certain set of global variables and DOM such as `Document` and `Cache`.
  2. While enforcing this constraint may seem easy, there are a number of subtleties related to the expressiveness and complexity of JavaScript. Common isolation techniques include blacklisting certain properties, separating the name-spaces corresponding to code in different trust domains, inserting runtime checks to prevent illegal accesses, and wrapping sensitive objects to limit their accessibility.

  This paper is similar to Caja [Mark S. Miller, 2007]. It studies how to combine runtime checks with syntactic restrictions that leads to secure subsets of JavaScript. It proposes two secure subsets of JavaScript that enforce isolation by means of synaptic restriction such as disallowing identifiers `eval`, `Function`, `Constructor`, `hasOwnProperty`. And it proposes three more semantic JavaScript subset that enforce runtime checks such as rewriting every occurrence of `this` in the code into the expression `NOGLOBAL(this)`.

- **Rewriting-based Dynamic Information Flow for JavaScript** [Dongseok Jang, 2010]

  Classification:

  Data-flow analysis.

  Description:

  This paper proposes a solution that is similar to Spectator [Livshits and Cui, 2008], paper[Jang et al., 2010], and paper [Vogt et al., 2007]. The solution it proposes to mitigate untrusted third party code (described here in Subsection 2.3.1) is the enforcement of

policies. For example, the object-location `(document; ''location'')` describes the object-location corresponding to the URL of loaded JavaScript code. An integrity policy is a map from object-locations to URLs whose code is allowed to influence the values stored at that object-location. The policies are enforced via a three-step process: taint injection, taint propagation and taint checking. The only difference between this paper and other similar papers is that this paper taints different JavaScript API and variables.

- **Preventing capability leaks in secure JavaScript subsets** [Finifter et al., 2010]

  Classification:

  Static analysis, Prevention.

  Description:

  This paper proposes a subset of JavaScript to prevent capability leaks.

  This secure JavaScript subset uses statically verified containment to prevent guests from using three classes of JavaScript language features, which if left unchecked, an attack could use these language features to escalate its privileges and interfere with the host page.

  - Global variables. This paper prevents third party code from reading or writing global variables. In particular, it require that all variables are declared before they are used (to prevent unbound variables from referring to the global scope) and forbid obtaining a pointer to the global object (to prevent accessing global variables as properties of the global object). For example, it bans the `this` keyword, which can refer to the global object in some contexts.
  - Dangerous properties. Even without access to global variables, third party code might be able to interfere with the host page using a number of special properties of objects. For example, if the third party code were able to access the constructor property of objects, it is possible to manipulate the constructors used by the host page. This paper implements this restriction by blacklisting a set of known-dangerous property names.
  - Unverified constructs. Because dynamically generated JavaScript code cannot be verified statically, this paper also bans language constructs, such as `eval()` that run dynamic script. In addition to dynamic code, this paper also bans dynamic property access via the subscript operator (e.g., `foo[bar]`) because bar might contain a dangerous property name at run time.

  Compared to ADsafe, which only blacklist known attacks, they prove that ADsafe is not perfect by monitoring the points-to relationship of objects in heap. Finally, they come to conclusion that instead of blacklist known attacks, it is better to only white-list known safe properties in a secure subset of JavaScript.

## 3.7 Drive-by download

Drive-by download is a common goal for JavaScript-based attacking. Since it requires a downloading behaviour, dynamic analysis technique is used to mitigate drive-by download. In this section, we have 5 papers to describe how to mitigate drive-by download.

- **Defending Browsers against Drive-by Downloads: Mitigating Heap-Spraying Code Injection Attacks** [Egele et al., 2009]

Classification:

Dynamic analysis (instrumented browser).

Description:

In this paper, a proof-of-concept implementation of a system that detects shell-code based drive-by download attacks is presented. The idea is to check the variables (strings) that are allocated by the browser (the script engine) when executing JavaScript. The detection is integrated into the browser, and performed before control is transferred to the shell-code, thus, effectively thwarting the attack.

The major tasks achieved are:

– Tracking object (String) allocation:
  More precisely, they added code to all points in the interpreter where string variables are created. These points were found at three locations: one for the allocation of global string variables, one for local string variables, and one for strings that are properties (members) of objects. The code added keeps track of the start address of a new string variable and its length.
– Checking strings for shell-code:
  They use Libemu to do the shell-code detection. If you are interested in libemu, please refer to `http://libemu.mwcollect.org/`.
– Performance optimizations:
  To reduce the performance penalty that is incurred when checking every string that is allocated, two techniques are implemented: First, one can reduce the total number of invocations of the emulation engine. While executing JavaScript core functionality, a script is allowed to create string objects without checks, even ones that contain shell-code. Second, one can reduce the amount of data that the emulator needs to inspect, by only recording information on all created string objects, and postpone emulation to the time at which control flow leaves the interpreter, entering an external component or the browser. The approach supports techniques to leverage speed-ups from both of these techniques.

- **Preventing drive-by download via inter-module communication monitoring** [Song et al., 2010]

Classification:

Dynamic analysis.

Description:

This paper proposes a dynamic analysis solution to monitor inter-module (e.g., basic HTML parsing, rendering engine, diverse plug-ins) communication (IMC) for detecting and preventing known drive-by download attacks.

The proposed detection mechanism works as:

1. Monitoring the communications to the vulnerable modules (e.g., Adobe Flash Player, Adobe Reader Plug-in) during a browsing session. They define three kinds of events to stand for the whole procedure of IMC:
   (a) Object creation. A creation of a component object indicates the beginning of a new communication session.
   (b) Method invocation. Invocations of methods constitute the main part of the communication.

   (c) Object free. The free of a component object indicates the end of the session.

2. Checking the communication content to identify known attacks, for improving the detect precision; here they use vulnerability-based signatures instead of traditional attack-based signatures. In more detail, they use symbolic constraint signature introduced in [Brumley et al., 2006].

- **IceShield: detection and mitigation of malicious websites with a frozen DOM** [Heiderich et al., 2011]

  Classification:

  Dynamic analysis, Prevention.

  Description:

  This paper proposes a dynamic analysis solution called in-line code analysis. In-line code analysis does the de-obfuscation first to get the unfolded JavaScript code. Then based on the behaviour of the code execution to decide whether it is a piece of malicious JavaScript or not. Totally, there are 8 heuristics implemented by IceShield:

  1. External domain injection: A script injects an external domain into an existing HTML element, which can indicate malicious activity, for example, link or form hijacking. They distinguish between injection of `<embed>`, `<object>`, `<applet>`, and `<script>` tags, as well as, `<iframe>` injections.
  2. Dangerous MIME type injection: A script applies a MIME type that is potentially dangerous to an existing DOM object such as `application/java-deployment-toolkit`.
  3. Suspicious Unicode characters: A string used as argument for a native method containing characters indicating a code execution attempt.
  4. Suspicious decoding results: Decoding functions like `unescape()` or `decodeURIComponent()` that contain suspicious characters indicating code execution attempts.
  5. Over-long decoding results: A decoding function like mentioned above receives an over-long argument. For now, they use a threshold of 4,096 characters based on our empirical evaluation of current attacks and benign sites.
  6. Dangerous element creation: A script attempts to create an element that is often used in malicious contexts for example, `<iframe>`, `<script>`, `<applet>` or similar elements. They distinguish between elements being created with and without an explicit name-space context.
  7. URI/CLSID pattern in attribute setter : An element attribute is being applied with an external URI, data/JavaScript URI or a Class ID (CLSID) string.
  8. Dangerous tag injection via the `innerHTML` property: A script attempts to set an existing element's value with a string containing dangerous HTML elements such as `<iframe>`, `<object>`, `<script>`, or `<applet>`.

  The most innovative work they do is the freezing of the document object. To freeze the document object, it is necessary to overwrite and wrap the native DOM methods into a context that allows researchers to dynamically inspect the name of the called function and its parameters during runtime. It is also possible to protect clients by overwriting the suspicious argument with an empty string or add randomly dimensioned padding to maliciously looking strings before passing them to the actual method.

  IceShield can be neutralized in case an attacker deploys a malicious PDF, Java Applet, or Flash without using any native DOM methods, and IceShield lacks of heuristics to cover all attacks (e.g., ActiveX based attacks).

- **Detection and analysis of drive-by-download attacks and malicious JavaScript code** [Cova et al., 2010]

  Classification:

  Hybrid.analysis.

  Description:

  In this paper Wepawet is introduced. The idea is to use machine learning to build a model of benign JavaScript code, such that when a new sample of JavaScript code does not fit the model, it will be classified as malicious. This paper introduces features that capture the following events:

  – Redirection and cloaking
  – De-obfuscation
  – Environment preparation
  – Exploitation

  Totally there are 10 features selected by Wepawet:

  1. Number and target of redirections
  2. Browser personality and history-based differences
  3. Ratio of string definitions and string uses
  4. Number of dynamic code executions
  5. Length of dynamically evaluated code
  6. Number of bytes allocated through string operations
  7. Number of likely shell-code strings
  8. Number of instantiated components
  9. Values of attributes and parameters in method calls
  10. Sequences of method calls

  All these features are based on the study of different attacks and should be able to capture the behaviour or effect of the attacks. These features require the usage of both static features and dynamic features.

- **Cujo: efficient detection and prevention of drive-by-download attacks** [Rieck et al., 2010]

  Classification:

  Hybrid analysis.

  Description:

  Fig. 3.11 shows the schematic description of Cujo. Cujo uses both static analysis and dynamic analysis. The analysis they use are existing technologies, which are static lexical analysis and ADSandbox [Dewald et al., 2010] dynamic analysis. After each analysis, a report is generated. We provide an example of the reports in Fig. 3.12. A big difference between Wepawet and Cujo is that Cujo does one more step: feature extraction, which Wepawet does not do. The feature extraction builds on the concept of q-grams, which has been widely studied in the field of intrusion detection. To unify the representation of static and dynamic analysis, Cujo first partitions each report into a sequence of words using white-space characters. Then a fixed-length window is moved over each report and extracts subsequence of q words at each position, so-called q-grams. For machine learning, Cujo uses SVM on q-gram.

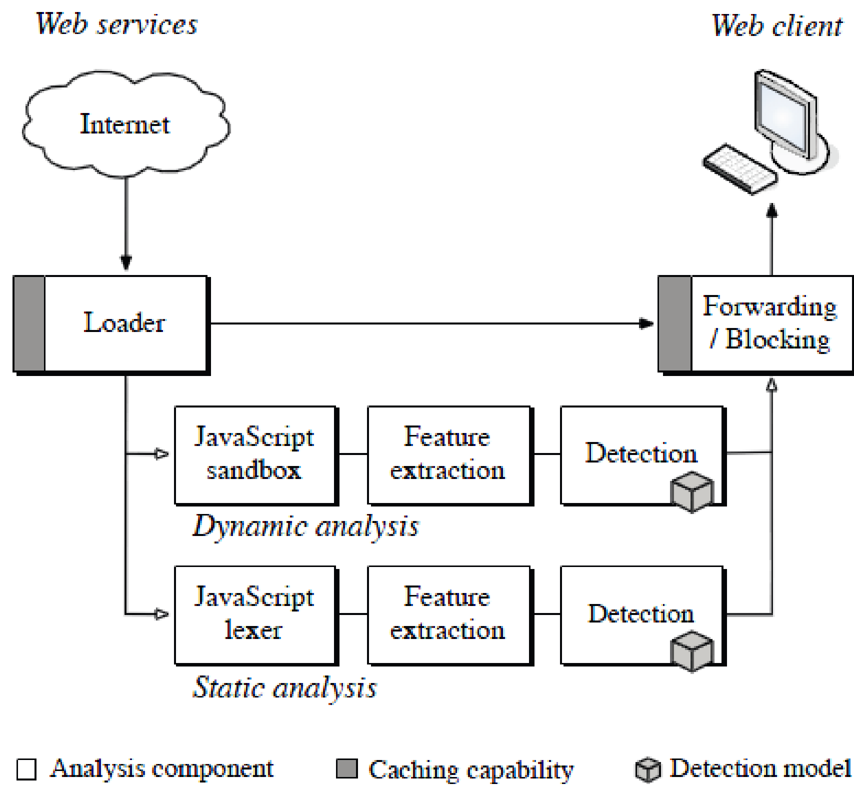


Figure 3.11: Schematic description of Cujo. Source: [Rieck et al., 2010].

```
1 ID = STR.00 ;
2 ID = STR.02 ;
3 FOR ( ID = NUM ; ID < ID . ID ; ID ++ ) {
4     ID = ID . ID ( ID ) - NUM ;
5     ID + = ID . ID ( ID ) ;
6 }
7 EVAL ( ID ) ;
```

```
1  SET  global.a TO ""
2  SET  global.b TO "{@xqhvfdsh+%(x<3<3%,>zkloh
3                    +{1ohqjwk?4333,{.@{>"
4  SET  global.i TO "0"
5  CALL charCodeAt
6  SET  global.c TO "120"
7  CALL fromCharCode
8  SET  global.a TO "x"
9  ...
10 SET  global.a TO "x=unescape("%u9090");
11                   while(x.length<1000)x+=x;"
12 SET  global.i TO "46"
13 CALL eval
14 CALL unescape
15 SET  global.x TO "<90><90>"
16 SET  global.x TO "<90><90><90><90>"
17 ...
18 SET  global.x TO "<90> ... 1024 bytes ... <90>"
```

Figure 3.12: Example of reports generated by Cujo. Up: Report of static analysis. Bottom: Report of dynamic analysis. Source: [Rieck et al., 2010].

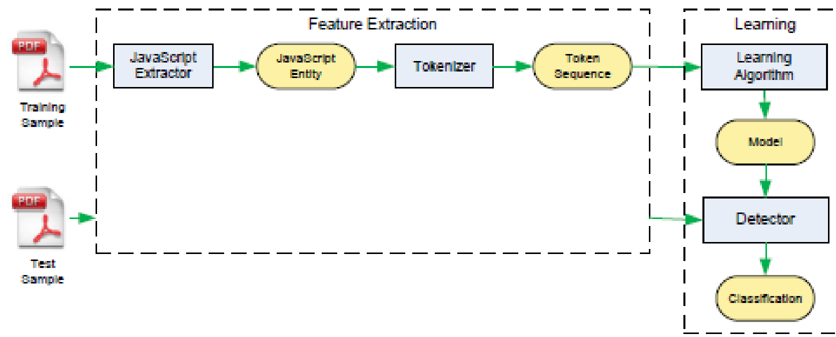


Figure 3.13: Architecture of PJScan. Source: [Laskov and Srndic, 2011].

## 3.8 PDFs with embedded JavaScript

In this section, we have 2 papers mitigating PDFs with embedded JavaScript. To extract JavaScript from PDF, static analysis technique is used.

- **Static detection of malicious JavaScript-bearing PDF documents** [Laskov and Srndic, 2011]

  Classification:

  Static analysis.

  Description:

  This paper proposes a tool named PJscan 3.13, which targets the detection of malicious JavaScript embedded inside PDF document. PJscan is closely related to static analysis techniques for detection of browser-based JavaScript attacks. The methodology is based on lexical analysis of JavaScript code and uses machine learning to automatically construct models from available data for subsequent classification of new data. The difficulty is the extraction of JavaScript code from PDF documents. Not only because PDF is a very complex format, but also PDF is rich with features that can be used for hiding the presence of JavaScript code. For example, it supports compression of arbitrary objects as well as various encodings for the JavaScript content.

  This paper studies the complex structure of PDF format, and indicates that JavaScript code can be found at the following locations of the PDF object hierarchy:

  1. After keyword `/js`
  2. In catalog dictionary's `/AA` entry
  3. In catalog dictionary's `/OpenAction` entry
  4. In document's name tree
  5. In document's Outline hierarchy

  After the extraction of JavaScript code, static lexical analysis follows. The analysis is based on the tokens as shown in Fig. 3.14. After this step, JavaScript code is tokenized. Based on these tokens, machine learning can be applied on and build the model for future detection.

- **Combining static and dynamic analysis for the detection of malicious documents** [Tzermias et al., 2011]

  Classification:

  Hybrid analysis.

| Value | Symbolic name | Description |
|---|---|---|
| 29 | TOK_NAME | identifier |
| 27 | TOK_LP | left parenthesis |
| 31 | TOK_STRING | string constant |
| 15 | TOK_PLUS | plus |
| 31 | TOK_STRING | string constant |
| 28 | TOK_RP | right parenthesis |
| 2 | TOK_SEMI | semicolon |
| 29 | TOK_NAME | identifier |
| 27 | TOK_LP | left parenthesis |
| 31 | TOK_STRING | string constant |
| 15 | TOK_PLUS | plus |
| 29 | TOK_NAME | identifier |
| 27 | TOK_LP | left parenthesis |
| 31 | TOK_STRING | string constant |
| 28 | TOK_RP | right parenthesis |
| 15 | TOK_PLUS | plus |
| 31 | TOK_STRING | string constant |
| 28 | TOK_RP | right parenthesis |
| 2 | TOK_SEMI | semicolon |
| 0 | TOK_EOF | end of file |

Figure 3.14: Tokens' definition in PJscan. Source: [Laskov and Srndic, 2011].

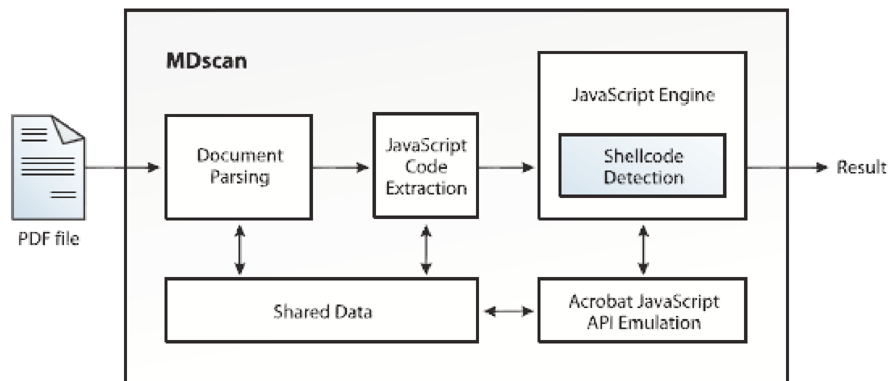


Figure 3.15: Overall architecture of MDscan. Source: [Tzermias et al., 2011].

Description:

The tool proposed in this paper is MDscan, which aims at analysing PDFs to protect it from malicious JavaScript. PDF scanning in MDscan consists mainly of two phases. In the first phase, MDscan analyses the input file and reconstructs the logical structure of the document by extracting all identified PDF objects, including objects that contain JavaScript code. The extraction of JavaScript code from the objects is achieved by searching codes after the keywords such as `/JS`, `/OpenAction` and `/AA`. In the second phase, any JavaScript code found in the document is executed by an instrumented JavaScript interpreter, which at runtime can detect the presence of embedded shell-code using Nemu[Polychronakis et al., 2010]. The overall design of MDscan is presented in Fig. 3.15

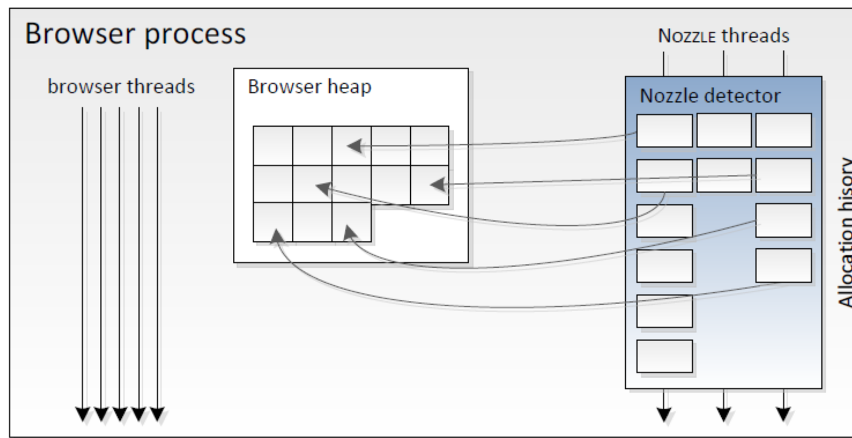


Figure 3.16: NOZZLE system architecture. Source: [Ratanaworabhan et al., 2009].

## 3.9 Heap spraying

As explained in Subsection 2.3.1, Heap spraying spreads NOP sled in heap. To detect this spreading behaviour and shell-code that is injected into the heap, a dynamic analysis component is essential. We describe a very important paper that specifically target this **vector**.

- **NOZZLE: a defence against heap-spraying code injection attacks** [Ratanaworabhan et al., 2009]

  Classification:

  Hybrid analysis.

  Description:

  This paper propose a tool named NOZZLE. Its architecture is shown in Fig. 3.16. NOZZLE is a two-level approach to detect heap spraying attacks: scanning objects locally while at the same time maintaining heap health metrics globally. At the individual object level, NOZZLE performs lightweight interpretation of heap-allocated objects, treating them as though they were code. This allows recognizing potentially unsafe code by interpreting it within a safe environment, looking for malicious intent by static analysis. In detail, NOZZLE scans heap objects to identify valid x86 code sequences, disassembling the code and building a control flow graph. The analysis focuses on detecting NOP sleds. Unfortunately, the density of the x86 instruction set makes the contents of many objects look like executable code, and as a result, existing methods lead to high false positive rates. To solve this problem they have developed a novel approach to mitigate this problem using global heap health metrics, which effectively distinguishes benign allocation behaviour from malicious attacks.

  Although there are existing solutions for NOP sled detection such as [Akritidis et al., 2005] and [Toth and Kruegel, 2002], but with high false positive since the lack of object selection. NOZZLE attempts to discover objects in which control flow through the object (the NOP sled) frequently reaches the same basic block(s), the assumption being that an attacker wants to arrange it so that a random jump into the object will reach the shell-code with the greatest probability.

  Nozzle has the following limitations:

  - Time-of-check to Time-of-use

Because NOZZLE examines object contents only at specific times, this leads to a potential time-of-check to time-of-use vulnerability. An attacker aware that NOZZLE was being used could allocate a benign object, wait until NOZZLE scans it, and then rapidly change the object into a malicious one before executing the attack.

– Interpretation start offset

  NOZZLE interprets the contents of objects as instructions starting at offset zero in the object, which allows NOZZLE to detect the current generation of heap-spraying exploits. However, if attackers are aware that NOZZLE is being used, they could arrange to fool NOZZLE by inserting junk bytes at the start of objects.

– Threshold setting

  The success of heap spraying is directly proportional to the density of dangerous objects in the program heap, which is approximated by NOZZLE. Increasing the number of sprayed malicious objects increases the attackers likelihood of success, but at the same time, more sprayed objects will increase the likelihood that NOZZLE will detect the attack.

– Targeted jumps into pages

  One approach to circumventing NOZZLE detection is for the attacker to eliminate the large NOP sled that heap sprays typically use. This may be accomplished by allocating page-size chunks of memory (or multiples thereof) and placing the shell-code at fixed offsets on every page.

– Confusing control flow patterns

  NOZZLE attempts to find basic blocks that act as sinks for random jumps into objects. One approach that will confuse NOZZLE is to include a large number of copies of shell-code in an object such that no one of them has a high surface area.

# Chapter 4

# Conclusions

In this chapter, we will indicate the next possible JavaScript-based threats and we suggest possible ways to mitigate them in Section 4.1. Moreover, we draw some conclusions for the whole thesis in Section 4.2

## 4.1 Future research directions

By studying the referenced papers, we find that from 2010, researchers start to realize that there is a common limitation for their mitigation approaches such as described in Rozzle [Kolbitsch et al., 2012], Wepawet [Cova et al., 2010], they are weak against environment specific attacks. If the mitigation approach/tool itself gets detected by malicious JavaScript code, then malicious JavaScript code can choose not to reveal itself or launch different attacks. For example, a piece of malicious JavaScript code can try to detect if NOZZLE [Ratanaworabhan et al., 2009] exists in the customer's machine as first step during code execution, then by simply using an `if` branch, if NOZZLE exists, malicious JavaScript code does nothing, if no NOZZLE exists, malicious JavaScript code can execute a piece of code that does heap spraying. There is a class of **evasion** techniques called environment matching techniques (described in Subsection 2.3.1) to detect existing mitigation approach or clients' environment. Nowadays, there are 2 techniques that can achieve such goal: fingerprinting and cloaking. We find that it is a reasonable trend for attackers, when it becomes more and more difficult to invent new **vector** or **technique** to attack.

Facing this new JavaScript-based threat, we think there are two ways that can be considered to mitigate it:

1. How to hide the existence of mitigation tool itself. For example, to implement mitigation tools such as Gatekeeper [Guarnieri and Livshits, 2009], it is necessary to modify the original JavaScript specification. As a consequence, to compile the code that suits the new specification, modification on browser interpreter is essential. But the modification on browser interpreter can be detect by JavaScript code when checking the browser information (e.g., browser version, available API name, global variable name) that is stored in the browser header and cookie. We think a sandbox solution can help to hide the truth of using a modified browser interpreter, the sandbox must implement browser environment mapping, which means wrapping the whole modified browser environment including JavaScript API, DOM and variables inside the sandbox, but from the outside the sandbox must appears like an unmodified browser, which means JavaScript code can use all original JavaScript API or variable without knowing that actually during code execution, another version of API or variables are running in the sandbox instead.

2. Detect and restrict the access to environment-related variables. Usually the environment variables such as browser information are stored into a header that is sent to the web server when a client do a web page request and these variables are backed-up into a cookie. By doing cookie tracking, environment variables can be taken illegally. We suggest one solution that takes two steps: (1) We give different privileges to browser interpreter, trusted JavaScript code and untrusted JavaScript code. (2) apply access-control policy to restrict the access to cookie. For example, untrusted JavaScript code can never read cookie.

## 4.2 Conclusions

In this systematization of knowledge thesis we present the peculiarities of JavaScript language and why JavaScript is easy to be abused, but difficult to be detected. We provide two timelines to show the evolution of JavaScript-based threats and mitigation techniques. We define new keywords: **vector**, **evasion** and **technique** to classify different JavaScript-based threats.

1. **Vector** means the methods to inject and execute malicious JavaScript code.
2. **evasion** means the techniques to evade existing mitigation approaches.
3. **technique** refers to the techniques used to exploit browser vulnerabilities.

We provide a global, high level taxonomy of mitigation approaches in which we classify mitigation approaches into Detection approaches and Prevention approaches. There are 3 subclasses of Detection approaches:

1. approaches using static analysis, which does not require code execution.
2. approaches using dynamic analysis, which requires code execution.
3. approaches using hybrid analysis, which combines static analysis and dynamic analysis and prevention.

We also provide one lower-level taxonomy for static analysis and another lower-level taxonomy for dynamic analysis. There are three subclasses of static analysis:

1. type analysis
2. AST-based analysis
3. points-to analysis

Meanwhile there are also three kinds of dynamic analysis:

1. VM-based
2. Sandbox
3. Instrumented browser-based

Especially, we find there is a hybrid analysis technique that is used by many approaches named data-flow analysis. We not only compare the differences between static analysis and dynamic analysis at high level, but also we compare the differences among mitigation techniques inside each class. We found some relationships between JavaScript-base threats and mitigation approaches such as:

1. defining new subset of JavaScript (one prevention technique) is effective against mash-ups (one **vector**).

2. data-flow analysis (a hybrid analysis technique) is effective against JavaScript extensions (one **vector**).

We describe each mitigation approach in detail and compare the similarities and differences among them to prove our knowledge. We think this thesis can help readers to have a view for JavaScript security, and can help readers to choose the class of mitigation techniques when facing a specific JavaScript-based threat. This thesis warns the readers to prepare for the next possible attacks and indicate the possible directions for further researches.

# Bibliography

P. Akritidis, E. P. Markatos, M. Polychronakis, and K. Anagnostakis. Stride: Polymorphic sled detection through instruction sequence analysis. In *In 20th IFIP International Information Security Conference*, 2005.

Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for javascript. In *Proceedings of the 19th European conference on Object-Oriented Programming*, ECOOP'05, pages 428–452, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-27992-X, 978-3-540-27992-1.

Davide Balzarotti, Marco Cova, Viktoria V. Felmetsger, and Giovanni Vigna. Multi-module vulnerability analysis of web-based applications. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 25–35, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-703-2.

Sruthi Bandhakavi, Samuel T. King, P. Madhusudan, and Marianne Winslett. Vex: vetting browser extensions for security vulnerabilities. In *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10, pages 22–22, Berkeley, CA, USA, 2010. USENIX Association. ISBN 888-7-6666-5555-4.

Adam Barth, Collin Jackson, and John C. Mitchell. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 75–88, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-810-7.

Adam Barth, Joel Weinberger, and Dawn Song. Cross-origin javascript capability leaks: detection, exploitation, and defense. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 187–198, Berkeley, CA, USA, 2009. USENIX Association.

David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP '06, pages 2–16, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2574-1.

Davide Canali, Marco Cova, Giovanni Vigna, and Christopher Kruegel. Prophiler: a fast filter for the large-scale detection of malicious web pages. In *Proceedings of the 20th international conference on World wide web*, WWW '11, pages 197–206, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0632-4.

Y. Cao, V. Yegneswaran, P. Possas, and Y. Chen. Pathcutter: Severing the self-propagation path of xss javascript worms in social web networks. In *Symposium on Network and Distributed System Security (NDSS)*, San Diego, California, USA, 2012.

Ramon Steenson Christian Seifert. Capture-hpc. 2006. URL `https://projects.honeynet.org/capture-hpc`.

Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for javascript. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 50–62, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1.

Marco Cova, Christopher Kruegel, and Giovanni Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 281–290, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-799-8.

Charlie Curtsinger, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. Zozzle: fast and precise in-browser javascript malware detection. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 3–3, Berkeley, CA, USA, 2011. USENIX Association.

Andreas Dewald, Thorsten Holz, and Felix C. Freiling. Adsandbox: sandboxing javascript to fight malicious websites. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 1859–1864, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-639-7.

Mohan Dhawan and Vinod Ganapathy. Analyzing information flow in javascript-based browser extensions. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, pages 382–391, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3919-5.

Vladan Djeric and Ashvin Goel. Securing script-based extensibility in web browsers. In *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10, pages 23–23, Berkeley, CA, USA, 2010. USENIX Association. ISBN 888-7-6666-5555-4.

Sorin Lerner Dongseok Jang, Ranjit Jhala. Rewriting-based dynamic information flow for javascript. 2010.

International ECMA. Ecma-262. June 2011.

Manuel Egele, Peter Wurzinger, Christopher Kruegel, and Engin Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Proceedings of the 6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '09, pages 88–106, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-02917-2.

Facebook. Fbjs. 2012. URL `http://developers.facebook.com/docs/fbjs/`.

M. Finifter, J. Weinberger, and A. Barth. Preventing capability leaks in secure javascript subsets. In *Proc. of Network and Distributed System Security Symposium*, volume 2010, 2010.

Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. *ACM Trans. Program. Lang. Syst.*, 21(2):370–416, March 1999. ISSN 0164-0925.

David Flanagan. *JavaScript: The Definitive Guide*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 3rd edition, 1998. ISBN 1565923928.

Salvatore Guarnieri and Benjamin Livshits. Gatekeeper: mostly static enforcement of security and reliability policies for javascript code. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 151–168, Berkeley, CA, USA, 2009. USENIX Association.

Oystein Hallaraker and Giovanni Vigna. Detecting malicious javascript code in mozilla. In *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, ICECCS '05, pages 85–94, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2284-X.

Mario Heiderich, Tilman Frosch, and Thorsten Holz. Iceshield: detection and mitigation of malicious websites with a frozen dom. In *Proceedings of the 14th international conference on Recent Advances in Intrusion Detection*, RAID'11, pages 281–300, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-23643-3.

Dongseok Jang and Kwang-Moo Choe. Points-to analysis for javascript. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 1930–1937, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-166-8.

Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. An empirical study of privacy-violating information flows in javascript web applications. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 270–283, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0245-6.

Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for javascript. In *Proceedings of the 16th International Symposium on Static Analysis*, SAS '09, pages 238–255, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-03236-3.

Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *Proceedings of the 2006 ACM symposium on Applied computing*, SAC '06, pages 330–337, New York, NY, USA, 2006. ACM. ISBN 1-59593-108-2.

John Kodumal and Alex Aiken. Banshee: a scalable constraint-based analysis toolkit. In *Proceedings of the 12th international conference on Static Analysis*, SAS'05, pages 218–234, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-28584-9, 978-3-540-28584-7.

Clemens Kolbitsch, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. Rozzle: De-cloaking internet malware. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 443–457, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4681-0.

Sophos Labs. Security threat report 2011. 2011.

Pavel Laskov and Nedim Srndic. Static detection of malicious javascript-bearing pdf documents. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 373–382, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0672-0.

P. Likarish, E. Jung, and I. Jo. Obfuscated malicious javascript detection using classification techniques. In *Malicious and Unwanted Software (MALWARE), 2009 4th International Conference on*, pages 47–54. IEEE, 2009.

Benjamin Livshits and Weidong Cui. Spectator: detection and containment of javascript worms. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 335–348, Berkeley, CA, USA, 2008. USENIX Association.

Gen Lu and Saumya Debray. Automatic simplification of obfuscated javascript code: A semantics-based approach. In *Proceedings of the 2012 IEEE Sixth International Conference on Software Security and Reliability*, SERE '12, pages 31–40, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4742-8.

Sergio Maffeis, John C. Mitchell, and Ankur Taly. Isolating javascript with filters, rewriting, and wrappers. In *Proceedings of the 14th European conference on Research in computer security*, ESORICS'09, pages 505–522, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 3-642-04443-3, 978-3-642-04443-4.

Sergio Maffeis, John C. Mitchell, and Ankur Taly. Object capabilities and isolation of untrusted web applications. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 125–140, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4035-1.

Ben Laurie Ihab Awad Mike Stay Mark S. Miller, Mike Samuel. Caja safe active content in sanitized javascript. 2007.

Microsoft. Microsoft web sandbox. 2012. URL `http://websandbox.livelabs.com/`.

Daniel Peck. Caffeine monkey. 2007.

Phu H. Phung, David Sands, and Andrey Chudnov. Lightweight self-protecting javascript. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, ASIACCS '09, pages 47–60, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-394-5.

Joe Gibbs Politz, Spiridon Aristides Eliopoulos, Arjun Guha, and Shriram Krishnamurthi. Adsafety: type-based verification of javascript sandboxing. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 12–12, Berkeley, CA, USA, 2011. USENIX Association.

Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos. Comprehensive shellcode detection using runtime heuristics. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 287–296, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0133-6.

Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin Zorn. Nozzle: a defense against heap-spraying code injection attacks. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 169–186, Berkeley, CA, USA, 2009. USENIX Association.

Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. Browsershield: vulnerability-driven filtering of dynamic html. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 61–74, Berkeley, CA, USA, 2006. USENIX Association. ISBN 1-931971-47-1.

Konrad Rieck, Tammo Krueger, and Andreas Dewald. Cujo: efficient detection and prevention of drive-by-download attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 31–39, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0133-6.

Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 513–528, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4035-1.

Ankur Taly Sergio Maffeis, John C. Mitchell. Run-time enforcement of secure javascript subsets. In *Proc of W2SP*, 2009.

Chengyu Song, Jianwei Zhuge, Xinhui Han, and Zhiyuan Ye. Preventing drive-by download via inter-module communication monitoring. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 124–134, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-936-7.

Ankur Taly, Úlfar Erlingsson, John C. Mitchell, Mark S. Miller, and Jasvir Nagra. Automated analysis of security-critical javascript apis. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 363–378, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4402-1.

Jeff Terrace, Stephen R. Beard, and Naga Praveen Kumar Katta. Javascript in javascript (js.js): sandboxing third-party scripts. In *Proceedings of the 3rd USENIX conference on Web Application Development*, WebApps'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

T. Toth and C. Kruegel. Accurate buffer overflow detection via abstract payload execution. 2002.

Zacharias Tzermias, Giorgos Sykiotakis, Michalis Polychronakis, and Evangelos P. Markatos. Combining static and dynamic analysis for the detection of malicious documents. In *Proceedings of the Fourth European Workshop on System Security*, EUROSEC '11, pages 4:1–4:6, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0613-3.

P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, San Diego, California, USA, 2007. NDSS.