POLITECNICO DI MILANO

Scuola di Ingegneria Industriale e dell'Informazione

Corso di Laurea Magistrale in Ingegneria Informatica

JavaScript threats: a systematization study

Relatore: Prof. Stefano ZANERO

Correlatore: Ing. Federico MAGGI

Tesi di Laurea Magistrale di:
Alessandro PRIMON
Matricola n.770732

Anno Accademico 2012/2013

Ringraziamenti

Al termine del lungo lavoro che mi ha portato alla realizzazione di questa tesi e alla conclusione dei miei studi è doveroso ringaziare coloro che mi sono stati vicini:

- **la mia famiglia** che mi ha sostenuto durante tutti questi miei anni al Politecnico, godendo dei momenti felici e sopportando con pazienza quelli meno piacevoli, spingendomi sempre a dare il massimo.
- **Emanuela** che ha reso questo ultimo periodo di studi più vivibile con la sua costante presenza e incoraggiamento...soprattutto leggendomi i voti degli esami!
- i miei **amici** che mi hanno regalato tanti bei momenti di svago e non hanno mai mancato di ricordarmi di dare il giusto peso alle cose, facendomi sempre capire quali sono quelle veramente importanti...tra le quali lo studio non rientrava!
- i miei colleghi **Simone** e **Davide** che hanno lavorato sodo per difendere la mia salute mentale dalle minacce del Politecnico. A loro il merito di avermi fatto passare questi sei anni con il sorriso sulle labbra.
- al **professor Zanero** e all'**ingegner Maggi** che mi hanno guidato durante tutto il percorso di ricerca e scrittura della tesi.

Contents

List of Figures

Listings

Sommario

JavaScript rappresenta l'attuale standard per la realizzazione di pagine web dinamiche e applicazioni web che forniscono un gran numero di funzionalità agli utenti, raggiungendo spesso una complessità paragonabile a quella di equivalenti applicazioni desktop. Negli ultimi anni questa tecnologia è stata ancor più direttamente integrata nei browser dando la possibilità di aumentarne le capacità installando estensioni realizzate appunto in JavaScript. Un'altra interessate evoluzione è data dall'adozione del linguaggio JavaScript per arricchire le potenzialità dei documenti PDF, che lo sfruttano per realizzare complessi rendering 3D, campi compilabili e altre funzionalità.

JavaScript nasce come linguaggio di supporto per la realizzazione di script lato client da parte di personale con limitate competenze di programmazione, mettendolo in condizione di realizzare una pagina web dinamica funzionante. Il linguaggio ha diverse caratteristiche, come la possibilità di generare codice durante l'esecuzione, il supporto a funzioni di prima classe e la gestione di oggetti che usano un paradigma di ereditarietà basato su catene di prototipi. Queste caratteristiche, che rendono l'analisi del comportamento di uno script un compito particolarmente impegnativo, hanno stimolato lo sviluppo di tecniche di attacco atte a compromettere l'integrità dei client che visitano una pagina infetta. La diffusione di attacchi perpetrati attraverso l'uso di codice JavaScript malevolo ha visto un'evoluzione nelle tecniche impiegate: si parte dagli attacchi di tipo cross-site scripting fino ad arrivare all'attuale fonte di minacce denominata drive-by download dove il codice JavaScript viene usato per sfruttare vulnerabilità presenti nel browser o in sue estensioni, con il fine di eseguire istruzioni arbitrarie sul computer client.

In letteratura è possibile trovare un gran numero di articoli che descrivono tecniche di difesa che contrastano le minacce provenienti da abusi di codice JavaScript malevolo. Una trattazione che espone una sistematizzazione di attacchi e relative contromisure non è ancora stata realizzata. L'obiettivo di questa tesi è sistematizzare lo stato dell'arte e i risultati della ricerca.

Con questa tesi si apportano i seguenti contributi:

1. Una descrizione dei meccanismi attualmente impiegati per la protezione dei client da attacchi effettuati sfruttando codice JavaScript.
2. Una tassonomia delle attuali tecniche di attacco.
3. Un'analisi passo-passo che mostra le singole fasi che portano alla realizzazione di un attacco, indicando le tecniche di difesa descritte nella letteratura che le possono contrastare e le aree che necessitano di maggiore copertura.
4. Una sistematizzazione che organizza e mette a confronto le tecniche di difesa descritte nella letteratura evidenziandone punti di forza e limitazioni.

Chapter 1

Introduction

JavaScript is a scripting language that has become the standard for the developing of dynamic web pages. It was originally designed to support scripts to enhance web pages user interaction, but today it is used to realize complex client side web applications that often offer the same features of an analogous desktop application. The employment of the language is so wide that today's web browsers allow to enrich their functionalities by integrating extensions written directly in JavaScript and also PDF documents can embed scripts to provide dynamic features like 3D rendering and fillable forms.

This dramatic increase in the use of JavaScript has attracted malware authors that started to employ it to perform their malicious actions. During the past 10 year is possible to identify a multitude of different attacks starting from cross-site scripting and then arriving to the emerging threat dictated by drive-by download attacks, where JavaScript code is employed to exploit vulnerabilities in the browser or in its extensions with the aim to execute arbitrary instructions on the client's machine.

In literature is possible to find a huge number of papers that describe defense techniques that counter the threats coming from the abuse of malicious JavaScript code. A single treatment that puts together a systematization of attacks and corresponding countermeasures is still missing. The objective of this thesis is to systematize the state of the art and the research results.

In this thesis these key contributions are presented:

1. A description of the current protection mechanisms employed for users' machines protection against attacks performed exploiting malicious JavaScript code.
2. A taxonomy of current attacks strategies.
3. A step-by-step analysis that shows the application of the defense techniques de-

scribed in literature for countering JavaScript based attacks and the areas that need further coverage.

4. A systematization that compares the defense approaches found in literature, showing their strengths and limitations.

Thesis organization.

- In chapter 2 are presented the characteristics of the language and the current protection mechanisms.
- In chapter 3 are described the attacks taxonomy and the step-by-step analysis that correlates attacks stages and corresponding mitigation strategies.
- In chapter 4 are organized the approaches built to detect attempts of set up attacks using JavaScript.
- In chapter 5 are organized the approaches designed to make the execution of JavaScript code inside the browser more secure.
- In chapter 6 are presented the conclusions and a description for possible future works.

Chapter 2

Background

2.1 Language characteristics

JavaScript is an interpreted language that gives developers an unusual freedom when writing their applications. These possibilities derive from a design specific that originally should allow personnel with limited programming skills to build up a working web page. These language features pose a great problem when dealing with JavaScript code in order to analyze its behavior. In the following are reported the most important characteristics of the language that will be used throughout the thesis.

Objects structure and prototypes. JavaScript objects doesn't have an immutable structure, they are treated as groups of properties, represented as records containing pairs of *name* and *value* that can be added or removed in complete freedom. The properties of an object are accessible both using the usual dot notation (like in Java) or specifying the property name as a string inside square brackets. JavaScript doesn't support the concept of object classes: objects inherit from other objects using a prototype based approach. This inheritance paradigm creates chains of dependencies represented through the property `__proto__`, which is assigned automatically to every object. The prototype chain is used when accessing property of an object: if the property name is not found in the current object, the prototype chain is navigated upward until the property is found or the top of the chain (represented by `Object`) is reached.

First class functions. As a design specific JavaScript supports first class functions i.e. allows to treat functions as values. In this way programmers can pass functions as parameters to other functions and return a function as result of a computation.

The support of first class functions hampers the applicability of static analysis making difficult to identify the real behavior.

Dynamic typing. JavaScript syntax doesn't enforce strong typing on variables, during the execution different types of values can be assigned to the same variable. To handle this feature dynamic checks are performed at runtime calling proper conversion routines when needed. This weak type system is often leveraged by malware authors to obscure their code.

Customizable execution environment and dynamic scoping. During web browsing the JavaScript code is interpreted by a component named JavaScript engine. For each page the engine provides a separate execution environment that contains all the basic blocks that support the execution: basic types, built-in objects and native functions. In addition to every relation created by the code all these basic blocks can be modified at runtime in order to change the overall behavior. This feature makes impossible to rely on a trustworthy infrastructure when applying security measures implemented directly in JavaScript. A further level of dynamism provided by JavaScript is the possibility to modify the scope of a block of statements. This is accomplished by defining an object that will be added to the scope chain through the use of the keyword `with`. In this case the object will be used to match properties used as unqualified names. Even without the use of `with` the analysis of the scope for a statement can be quite challenging being by default employed a function-level scope.

Dynamic generated code. JavaScript gives the possibility to include additional code at runtime. This task can be accomplished in different ways: (i) dynamic evaluating a string using `eval` or similar constructs, (ii) including a new `script` tag using the DOM APIs or (iii) importing a remote script . This feature is particularly used in anti-defense techniques as described in section §3.3.

2.2 Current protection mechanisms

2.2.1 Same-origin policy

The Same-origin policy (SOP) is the basic security measure enforced natively by browsers to provide confidentiality and integrity of data belonging to web pages. Its basic concept is to allow active content (e.g. JavaScript code and Flash objects) to access only resources coming from the same origin, where the origin is a record composed by (i) protocol,

(ii) hostname and (iii) port number . With the term resources is intended whatever object related to a specific page, be it the DOM tree, a JavaScript object, a CSS file, an image, a cookie etc. The presence of the SOP has become more and more important with the rise of complex web sites which make intensive use of HTTP cookies, because without a proper security measure the information stored inside them would be freely accessible by other pages. Another important aspect of the SOP is that it provides a minimal level of protection against undesired information flows disallowing XMLHttpRequests (XHR) to a server different from the one where the page comes from. The main limitation of the SOP lies in its inability to protect a page from the threats coming from included content that, despite its real source, is considered as belonging to the page origin and so able to access all the native content, including the sensitive information.

2.2.2 Content security policy

The content security policy[1] (CSP) is a protection mechanism that establishes a collaboration between server and browser to filter the content included in a web page. This security measure is built to secure web pages against attacks that try to embed malicious content through the specification of trusted content origins. In practice the server includes a special header in the response sent to the browser which in turn must enforce the specified policies. The header contains many fields that define a whitelist of origins used as source of different types of content like JavaScript, images, fonts, CSS files etc.; is also possible to specify restrictions on the protocol used to serve the content (e.g. accepting only scripts served using HTTPS). The implementation of such policies requires a considerable development effort in particular for web page designers that, together with a prudent definition of the policies, must reorganize their pages in order to remove all the inlined elements, moving them in separate files, making possible to enforce effectively the policies. In addition to its whitelisting capabilities the CSP allows for a coarse grained control over the execution of scripts, for example is possible to block the use of `eval` or the execution of inlined scripts, which may be the result of a successful stored XSS attack. The limitation of the CSP lies in its limited power in exerting control over the imported scripts.

2.2.3 Antivirus systems

Signature matching detection. The current state-of-the-art protection provided by commercial antivirus systems is based on the matching of signatures representing known

[1]Content Security Policy 1.0, `http://www.w3.org/TR/CSP/`

malware families. This protection mechanism performs a sole static analysis of all the JavaScript code retrieved when a web page is visited. Unfortunately this approach has the following limitations:

- the antivirus is always one step behind the attacker because every novel attack needs an ad-hoc signature to be identified. This issue leaves open a vulnerability window, from the attack release to the reception of the update by the antivirus.
- using simple obfuscation techniques (described in section §3.3.2) an attacker can evade the detection by radically modifying the shape of its code, making it no more compliant to the defined signature. The variety of obfuscation strategies available for an attacker makes the creation of a complete set of signatures practically unfeasible.

Downloaded files scanning. A feature available by almost all antivirus system is the possibility to scan the files downloaded by the web browser. This methodology provides a basic defense against drive-by download attacks that relies on the download of malicious binaries to infect the victim machine. Consideration on the effectiveness of this security measure are analogous to those of signature matching detection.

Reputation systems. A novel approach implemented to add a further level of protection during web browsing is the possibility to block the loading of a page that has a "bad reputation". The idea of this approach is to consult a public database to check the reputation of a page and start the page retrieval only if the reputation is good. This protection is often provided by commercial antivirus systems integrating an extension in the web browsers installed in the client machine. The interesting aspect of this method is the possibility for the users to contribute in improving the reputation estimate of a page by submitting a personal opinion; in this way is also possible to create filters for parental control and identify illegal web pages e.g. child pornography distribution sites or drugs online shops. The basic limitation of this approach is the lack of protection for web pages for which a reputation estimate is not available and the possibility for attackers to bias the estimate submitting false opinions.

Chapter 3

JavaScript-based attacks

In this chapter is presented a detailed analysis of the structure of the common attacks perpetrated against web browsers with the use of malicious JavaScript code. An overview is presented in figure 3.1 where each white block represents a single step that contributes in realizing the attack. Blue and green boxes indicate the defense measures that have been developed to counter the attack acting on the contained basic blocks; orange boxes indicate a partial solution to the contained blocks. Finally red blocks represent successful attacks. Each of the following sections treats a particular aspect of the attack methodology and includes the explanation of the blocks belonging to the corresponding gray box in figure 3.1.

3.1 Attacks characterization

Information leakage. With the rise of interactive web applications the sensitive information stored and managed with web browsers are become and attractive target for malware authors. Using JavaScript an attacker can collect valuable information that are sent to a server under their control to perform more dangerous operations (like cross-site request forgery) or to be sold to third parties.

Control flow hijacking attacks. Traditional attacks that leverage memory errors in the browser can be performed also using JavaScript. Even though their potential harm is always notable, their employing is becoming less prevalent with the rise of drive-by download attacks.

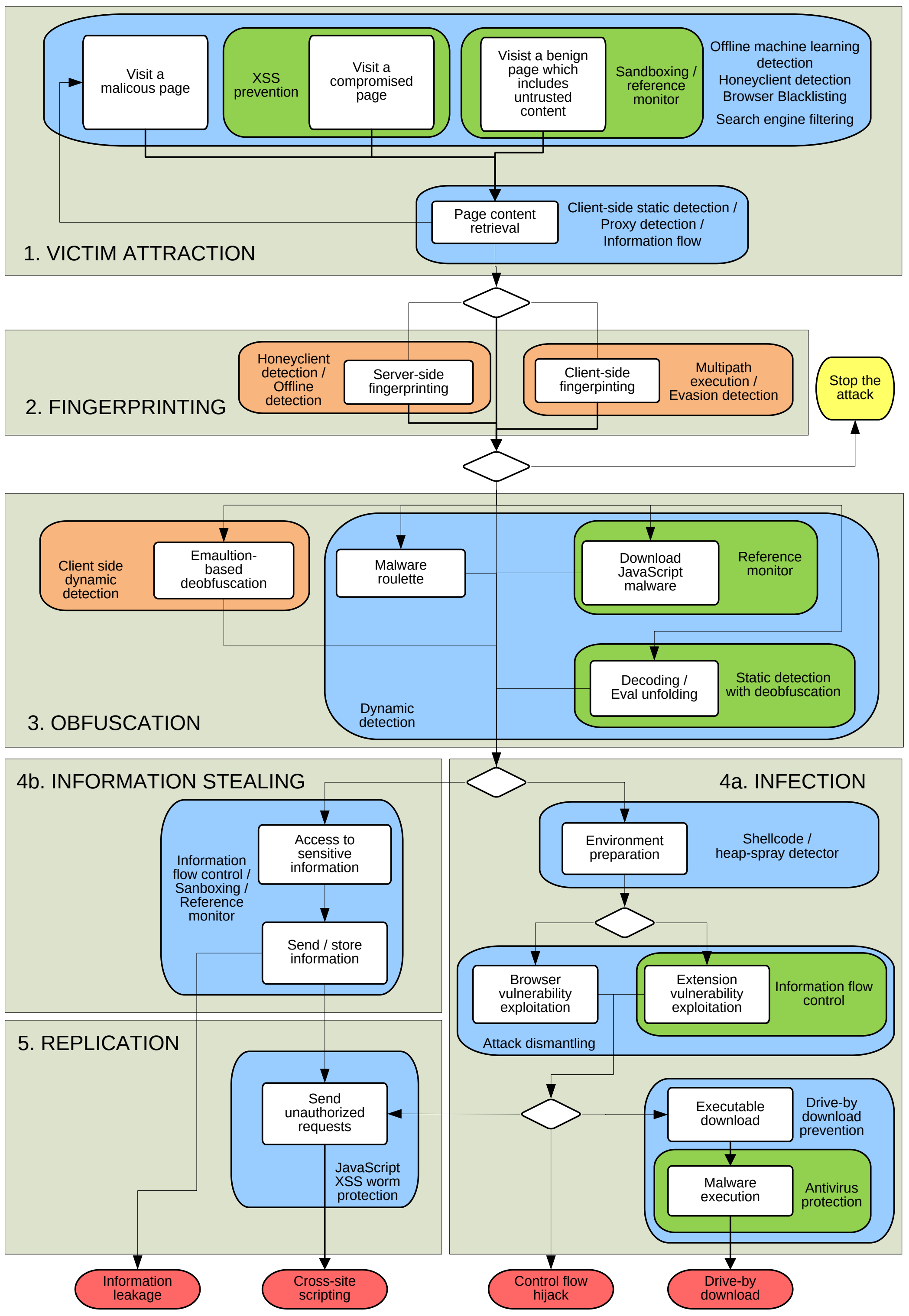
1. VICTIM ATTRACTION
Visit a malicous page
XSS prevention
Visit a compromised page
Visist a benign page which includes untrusted content
Sandboxing / reference monitor
Offline machine learning detection
Honeyclient detection
Browser Blacklisting
Search engine filtering
Page content retrieval
Client-side static detection / Proxy detection / Information flow
2. FINGERPRINTING
Honeyclient detection / Offline detection
Server-side fingerprinting
Client-side fingerpinting
Multipath execution / Evasion detection
Stop the attack
3. OBFUSCATION
Client side dynamic detection
Emaultion-based deobfuscation
Malware roulette
Download JavaScript malware
Reference monitor
Decoding / Eval unfolding
Static detection with deobfuscation
Dynamic detection
4b. INFORMATION STEALING
Information flow control / Sanboxing / Reference monitor
Access to sensitive information
Send / store information
4a. INFECTION
Environment preparation
Shellcode / heap-spray detector
Browser vulnerability exploitation
Extension vulnerability exploitation
Information flow control
Attack dismantling
5. REPLICATION
Send unauthorized requests
JavaScript XSS worm protection
Executable download
Drive-by download prevention
Malware execution
Antivirus protection
Information leakage
Cross-site scripting
Control flow hijack
Drive-by download

Figure 3.1: The anatomy of a modern JavaScript-based malware.

Drive-by download attacks. In a drive-by download attack the JavaScript code is employed to download and execute a malicious executable in order to infect the victim machine. The infection is usually made possible exploiting a vulnerability in the browser or in its components (e.g. ActiveX controls and Adobe Flash plug-in). Once the machine has been infected the attacker can gain a complete control over it installing rootkits and maybe making it join to a botnet. This class of attacks has gained more and more popularity in the recent years being employed by an increasing number of attackers, driving research community to an intense effort to build up proper defense strategies.

Cross site scripting and XSS worm. Once the malicious code has performed its malicious actions on the victim's machine it has the possibility to replicate itself on the hosting page to infect other visitors. This behavior is known as cross-site scripting (XSS) worm and is usually found in social networks, where the victim is infected visiting a friend's profile and after the infection the malware inserts itself on the victim's profile. The spreading of such a malicious code is made possible by the presence of XSS vulnerabilities in the web application. This emerging threat is quite worrying because a XSS vulnerability in a popular social network can rapidly cause the infection of millions of machines, as already happened in MySpace with the Samy worm[1].

3.2 Victim attraction and page retrieval

As shown in the block 1. of figure 3.1 the first step needed to perform an attack employing malicious JavaScript code is to make the user visit the web page where the malicious code is located. To attract users attackers can use different strategies:

- **Set up a malicious page:** in this case the attacker exposes the malware on a page under his control. To attract users he can try to include the page in the search engines results or insert links in more popular sites, maybe leveraging the obfuscation infrastructure provided by URL shortening services [2]. This technique is however quite easy to counter: once an offline analysis system like JSAND [3] or a honeyclient-based infrastructure (§ 4.7) detects the malware the page can be inserted in blacklists (like Google Safe Browsing) to be filtered by browsers and excluded by search engines results. Knowing that many pages are used as hub to keep links to malicious pages a tool like EvilSeed [4] can be used to improve the detection effectiveness.

[1]The Samy worm, http://namb.la/popular/

- **Compromise a benign page:** an attacker may try to include its malicious code by compromising a benign (an hopefully popular) web page, in order to have a wider victim pool and bypass blacklist defenses. To perform this task he can appeal both to web server vulnerabilities or cross-site scripting (XSS) vulnerabilities in the web application. In the case of malicious JavaScript injected via an XSS vulnerability the harm can be reduced by configuring a properly the CSP for the page (thus disabling the execution of inlined code) or relying on XSS prevention mechanism like BEEP [5] o BluePrint [6]. An additional problem in detecting a compromised benign page, that arises also in the case of malicious pages, is the habit of attackers to serve their malicious code only in certain periods: this fact make necessary repeated scans of the same page to detect the malware.

- **Unsafe mash-up:** today's web pages include third-party content to enrich their user experience delivering a number of different services: these pages are usually known as *mash-ups*. An attacker can leverage this programming practice by luring the web page developer in including a script that contains a malicious behavior that, not being subject to the same-origin policy, has full privileges to perform its actions. Depending on the nature and on the complexity of the content to include the developer can choose to adopt one of the defense measures described in sections § 5.1, § 5.2 and § 5.3.

If one of the previous scenarios occurs, leading to the browser retrieving the page and the malicious content, is still possible to protect the machine from being compromised. A viable solution is to interpose a proxy that performs an analysis of the page using a tool like CUJO [7], delivering the page only if no malware is detected; the alternative is to let the execution start relying henceforth on a detection tool like IceShield[8] or Zozzle[9]. A detail that can be taken into account is that the effect of visiting a compromised page or an unsafe mash-up can be the sole redirection to a malicious page: in this case the protection can be performed by an information flow control tool like [10] or [11] that denies the flow of untrusted data inside the `document.location` property.

Malicious PDF documents. The protection against malicious PDF documents, downloaded through the web browser or in any other way, can only rely on ad-hoc detection tools like the ones described in section §4.3, that are hopefully integrated in traditional antivirus systems.

3.3 Evasion techniques

Together with the implementation of defense techniques that counter attacks performed using JavaScript, attackers have devised ways to keep their attacks effective against user machines. Today build up a protection mechanism that doesn't take into account the presence of such evasion techniques will perform very poorly due to they wide adoption.

3.3.1 Fingerprinting

The term fingerprinting indicates those techniques which aims to detect the current system configuration where a script is executing. Even if the presence of such techniques is not symptom of malicious behavior, their utilization is not so common: as reported in [12] only a small fraction of JavaScript samples refer to environment dependent variables and even a smaller fraction uses those values in branches condition, by contrast the great majority of malicious scripts relies on environmental variables. The main motivations that lead attackers to implement fingerprinting techniques concern the enhancement of the attack success likelihood that in practice translates in (i) retrieve the correct script for a certain system, (ii) prepare a proper execution environment for the attack and (iii) hide the malicious behavior in presence of sandboxes, detection tools or honeyclients (cloacking) .

Fingerprinting techniques are usually implemented at client side because certain checks can be performed only in this way (e.g. verify the presence of a plug-in), however is possible to realize a limited form of fingerprinting also at server side. In this case the checks can involve only limited set of environmental parameter like the operating system, the browser version, the language and the IP address. Client side fingerprinting can be dismantled by an analysis system with different strategies like honeyclients (varying the configuration) and multiexecution tools like Rozzle [12], Revolver [13] and Kudzu [14].

Server side fingerprinting is instead more difficult to counter because it can be identified only with repeated visits to a page using different system configurations and possibly from different locations. The location issue is very important because attackers usually leverage lists of IP addresses of known analysis organizations to hide their malicious code. An attacker aware of the presence of any defensive measure can still try to evade detection by exploiting an intrinsic flaw that plagues most of the analysis and detection tools: the overhead posed by the analysis process allows an attacker to distinguish between protected and unprotected clients by measuring the execution time of certain operations. In many case the results are not so different from those of a slow machine but for heavy offline analysis tools, that can slowdown the execution by a factor of 100, the job for the attacker is unbelievably easy.

3.3.2 Anti-analysis techniques: defeating static analysis

The usage of obfuscation techniques on a piece of JavaScript causes the modification of its form while preserving the original behavior, the only difference between the two versions of the code is that the latter is almost not human readable. This processing stage is employed by attackers both to hamper the manual inspection performed by security researchers and to evade the detection of automated tools. The main concern when facing the problem of detecting malicious obfuscated code is that the presence of obfuscation is not an index of maliciousness: developers widely employ such techniques to protect the copyright of their code. Although JavaScript provides many facilities to realize an effective obfuscation, tools that automatize the process are available on the web and widely adopted. When dealing with the detection of obfuscated JavaScript code the basic concept to keep in mind is that dynamic analysis techniques can be effective as the malicious code will reveal itself at runtime in order to be executed. The effectiveness is enhanced employing techniques that detect malicious code basing on "how the code behaves" instead of "how the code looks like" (§ 4.1).

White spaces and comments randomization. The simplest approach to prevent the reading of a piece of JavaScript is the insertion of randomly generated white spaces and comments between keywords. This operation preserves the original semantics of the code because the JavaScript interpreter ignores white spaces and comments: in this case the deobfuscation is not delegated to the attacker. Although the deobfuscation is easy to be performed also by hand, a detection tool based on signature matching can be completely evaded because the shape of the code is dramatically different.

Minification. An interesting aspect is the possibility to employ the inverse approach with respect to white spaces randomization: the minification is the process that modifies the code in order to reduce its size and so the loading time. This technique is wide adopted both by benign and malicious developers that leverage the syntactic changes introduced to evade signature matching systems. The minification process involves the removal of white spaces and comments and also the modification of symbols, substituting a shorter alias when possible. Many tools for the minification of JavaScript code are available on the web[2][3][4].

Integer obfuscation. Another simple approach to modify the shape of the code keeping its behavior unaltered is the integer obfuscation which requires the representation of numbers as algebraic operations. Considerations made for white spaces randomization and minification holds in also in this case.

Strings obfuscation. The most common approach used by attackers to obfuscate a piece of JavaScript is to represent the code as a string and apply to it an encoding routine. At runtime is sufficient to apply the respective decoding routine and execute the resulting code passing the deobfuscated string to `eval`[5]. The obfuscated string can be divided in several chunks and then reassembled when needed or hidden inside the HTML markup and retrieved using the DOM API: this last technique is particularly effective in the case of simple detection systems that rely only on a JavaScript interpreter and thus cannot provide DOM API support. The encoding strategies natively supported by JavaScript are numerous (e.g. %-encoding, Unicode encoding, Base64 etc.) but a determined attacker can build up an ad-hoc decoding routine to further harden the obfuscation. In addition he has the full freedom to add several layers of obfuscation, hiding also parts of the decoding routine and showing the real malicious behavior only at the end of the process: this technique which requires repeated dynamic code evaluations is known as *eval unfolding*. The obfuscation techniques described so far are powerful to counter simple static analysis detectors like PROPHILER [19] and signature matching systems but a dynamic analysis tool [7, 8, 3], or a smarter static analysis tool [9] can extract the unobfuscated version of the code. As last resort an attacker can appeal to encryption, maybe using environment variables as key in order to deny the decryption by analysis tools that do not impersonate correctly a real browser environment.

```javascript
var percent = unescape("%64%6f%45%76%69%6c%28%29%3b");
var unicode = "\u0064\u006f\u0045\u0076\u0069\u006c\u0028\u0029\u003b";
var base64 = atob("ZG9FdmlsKCk7");
```

Listing 3.1: Examples of endcoding of the string "doEvil();"

```javascript
var sh = "ZG9FdmlsKCk7"; // Base64 encoding of doEvil();
var sh2 = "var sh1 = atob(sh); eval(sh1);";
```

[2]JSMin - The JavaScript Minifier, `http://www.crockford.com/javascript/jsmin.html`
[3]YUI Compressor, `http://yui.github.io/yuicompressor/`
[4]Dojo ShrinkSafe, `http://shrinksafe.dojotoolkit.org/`
[5]Other methods used in practice are `document.write()`, `element.innerHtml()` and `setTimeout()`

```
eval(sh2);
```

Listing 3.2: Example of eval unfolding

Property access with square bracket notation. JavaScript syntax allows one to access properties of an object both using the usual dot notation or the square bracket notation: the latter is widely employed by attackers to obfuscate their code. The interesting aspect is the possibility to access a property whose name is stored inside a string, in this way an attacker can hide the name of the property that he is going access using the already described techniques.

```
var sh = "<script> doEvil(); </script>";
var pro = 'w' + "\u0072\u0069" + atob("dGU="); // Obfuscated version of "
    write"
document[pro](sh); // Alternative way to call document.write();
```

Listing 3.3: Examples of access through square bracket notation

Variable name randomization and function pointer reassignment. To hamper the analysis based on the identification of sequences of native functions calls [20] an attacker can randomly generate a number of new variable names that are then associated to the desired functions. In this way a tool that simply searches for the name of the functions gets fooled. A smarter dynamic analysis tool can however bypass these obfuscation techniques by hooking the native functions, however some of the tools described in section §4.1 do not provide such feature.

```
randomName1 = atob;
randomName2 = eval;
var sh = randomName1("ZG9FdmlsKCk7"); // Base64 encoding of doEvil();
randomName2(sh);
```

Listing 3.4: Example of function pointer reassignment

Malware roulette. The malware roulette [12] is a technique based on the rewriting of a piece of JavaScript in order to hide as much code as possible thus avoiding detection. The basic idea is to split the original script in its basic blocks and retrieve the each basic block when needed sending a request to the server including a representation of the actual state of execution. In this way the server provides the least possible code making difficult for a static detector to correctly identify the attack, anyway this technique doesn't pose any problem for a dynamic detection tool that relies on the tracking of the operations performed by the script. Even if this technique aims to obscure the execution of malicious code to avoid detection it has a component of fingerprinting: together with the execution state the script can send also environmental information so that the server can provide the correct code for the specific system configuration.

Emulation-based obfuscation. The employment of emulation techniques to obscure JavaScript code is a big concern for detector developers. This technique requires the translation of the code in a stack-based bytecode that is interpreted at runtime by a dispatching routine which delegates the real operations to a set of ad-hoc functions. The problem in detecting such an obfuscation strategy is that the detector can see, in the best case possible, only the structure of the dispatching routine and the functions[6], while the bytecode is no more than a string. To further obscure the bytecode the attacker can hide its entry point, making difficult for an analyst to extract the real code. To hamper the analysis performed by an automatic tool the attacker can appeal to the technique named *implicit conditionals*, where the entry point is calculated with a function based on environmental variables without conditionals: in this way a tool that doesn't mimic correctly a real execution environment (also with respect to performance) can be fooled by showing a benign behavior. In [18] Lu and Debray demonstrated that the combination of these techniques can effectively evade the detection of state-of-the-art tools like JSAND [3] and ZOZZLE [9]: even if few countermeasures have been identified a general solution with a practical implementation is still missing; diversely the sole use of emulation can be effectively identified by JSAND. A possible approach that, to my knowledge, has never been adopted so far is to protect the browser using a tool like ICESHIELD [8] that performs lightweight protection detecting the malicious code on the base of its behavior.

3.3.3 Anti-debugging techniques: defeating dynamic analysis

Being aware of the wide employment of honeyclients by security researchers to study their JavaScript malware attackers developed fingerprinting techniques that frustrate this analysis approach. The main objective of such techniques is to keep hidden the malicious behavior but a more determined attacker may try to exploit vulnerabilities in the analysis system to make the attack invisible or even compromise the honeyclient.

Honeyclient detection. The most important characteristic that an honeyclient must possess is the ability to mimic correctly a real execution environment, in order to allow

[6]they both can be obfuscated with previous describe techniques, being included as string and deobfuscated at runtime

the JavaScript code to execute as expected. Unfortunately is difficult to fully agree to this specification and in practice certain differences can be found in every honeypot systems. These flaws are exploited by attackers to distinguish a legitimate system from an emulated one, with the purpose of leaving undetected their code. The easiest way for an attacker to detect an honeyclient is to verify if the browser is executing inside a virtual machine: this is a common practice followed by many researchers thank to the possibility of setting up automatically multiple virtual systems that could be compromised as result of a successful attack. There are many viable methods that can perform this check but the important aspect is that an effective countermeasure must be implemented or a significant fraction of analysis systems can become useless. Another possibility for the attacker is to detect the presence of a particular honeypot system by looking for artifacts that are left behind. Usually attackers lure the JavaScript engine in loading a file belonging to the honeypot as if it was a script file and, if the file is found, they stop any malicious attempt. Luckily this evasion technique can be countered effectively by instrumenting the browser not to load files that do not contain JavaScript code performing a check on their extension.

Detection evasion. The other choice for an attacker to evade detection is to make the malicious behavior invisible to the honeyclient: knowing the detection approach different techniques can be employed. Most of the honeyclient-based detection systems have a maximum execution time for a single web page in order to guarantee acceptable performance. An attacker can exploit this design specific delaying the execution of the malicious code after a period greater than the honeyclient timeout simply leveraging the `setTimeout()` function. Even if this technique is effective against a general honeyclient, a smarter tool can decide to continue the analysis of the page until all the active timers are terminated or devise a method to force their termination. Luckily setting large timeouts to evade the detection makes attacks unreliable because a legitimate user may not remain on the page enough time to trigger the execution of the malicious code. A similar approach requires the inclusion of the malicious code inside event handlers, that most of the honeyclient do not trigger, anyway this technique is even easier to counter, requiring the sole stimulation of the registered events as done in [22]. The standard detection approach taken by the majority of the honeyclient systems is to monitor the underlying operating system, reporting a possible attack when unexpected event occurs (e.g. files and registry entries creation). This methodology leaves the possibility to an attacker to carry out an attack whose operation have effects only in memory. An example is to exploit a vulnerability in the web browser, retrieve a remote library and create a separate thread in the browser process where the attack can continue its execution without being detected. Confining the attack in main memory prevents the execution of certain operations like the download of files, but still allows to steal credentials, produce network traffic and carry out other types of attacks like sending spam or conduct a denial of service attack.

3.4 Information stealing

Users' sensitive information are the target of many attacks performed using JavaScript because they are quite easy to obtain. Once the malicious script has been included in a web page (both by the developer or as result of a XSS attack) it has full privileges to access to the sensitive information here stored (e.g. cookies) and then send them over the network. To counter this attack scenario is possible to block the access to certain information by realizing confinement of third-party included scripts using a sandboxing framework (§ 5.1) or, to have a wider protection that applies also to scripts injected without the developer consent, implementing the policy enforcement defense provided by reference monitors (§ 5.2). The alternative is to block the undesired spreading of information outside the browser using a information flow control tool like [10] or [11].

3.5 Exploitation

One of the objectives of an attacker is to take control of the victim machine in order to perform more dangerous operations with respect to those allowed using JavaScript. In this case he must exploit vulnerabilities in the web browser or in its components to divert their control flow. The most common technique is to store a shellcode in memory and then try to execute it exploiting the aforementioned vulnerability, usually performed calling a vulnerable function passing a proper argument. This scenario is common in every attack that leverages a memory error and adequate countermeasures have already been adopted like address-space layout randomization. Attackers have anyway devised many ways to overcome these protection mechanisms, the most employed is the so-called heap-spraying where multiple instances of the shellcode are stored in the heap, enhancing the attack success likelihood. Given the memory allocation strategy employed by JavaScript the only reliable way for an attacker to store contiguous sequences of executable instructions in the heap is to to store them in a string. To counter this attack scenario researchers developed tools that inspect the heap at runtime looking for objects that can be treated as executable code and blocking their execution before the vulnerability is triggered (§ 4.2). A different approach is the employment of IceShield [8] that is able to dismantle the attack by modifying suspicious parameters value making the exploit fail.

Once the attacker diverted the control flow of the browser, he has full freedom to perform every malicious action of his choice, however most of the times this ability is employed to retrieve a remote malware binary, performing a drive-by download attack. An alternative way to download such a malicious executable is to leverage a vulnerable API provided by a JavaScript extension. In this case an attacker misuses the API performing legitimate function calls with properly designed parameters, that lead the extension to download and execute the remote file. To stop this misuse of browser extension tools like Sabre [23] and vex [24] have been developed to track the flow of information that can cause untrusted code execution with high privileges. If all the previous security measures failed in stopping the attack, is still possible to prevent the machine from being compromised, relying on tools specialized to counter this threat (§ 4.6).

3.6 Replication

After its malicious actions have been performed a malicious script can try to replicate itself using a XSS vulnerability of the hosting web application, showing a behavior identifiable as a JavaScript worm. If a collaboration tool like beep [5] or BluePrint [6] failed in stopping the execution of the injected script is possible to block the spreading of the malware using specialized tools like PathCutter [25] or Spectator [26] that require the intervention of the server.

Chapter 4

Malicious JavaScript detection

This chapter describes those tools developed to detect the presence of malicious JavaScript code inside a web page. The main problem that these tools face is to provide an effective protection generating the least possible number of false alarms and without adding a considerable overhead to the page visualization process: this in turn can lead to the tool being removed from the browser by the producer or disabled by the user, leaving him unprotected from the threats coming from the visited web pages. In figure 4.1 is reported the global taxonomy of the defense approaches described in this chapter, organized both by technique and application field (represented as the blue blocks).

4.1 Machine learning-based malware detection

The tools that fall in this category try to classify the page as benign or malicious basing on features extracted from the JavaScript code and the HTML content. The features employed in the classification process can be both retrieved statically or dynamically but, due to the wide adoption of obfuscation strategies by malware authors (§ 3.3.2), the sole use of static features can leave undetected a large portion of the malicious code; unfortunately the use of dynamic features involves the execution of the JavaScript code, causing a limitation in the code coverage and dramatically increasing the time required for the page analysis and giving the attacker a chance to exploit the analysis system.

A first example of such tool is CUJO [7] which analyzes pages inside a proxy, providing them to the user only if they are classified as benign. At each request CUJO contacts the server and retrieves the desired page, all the included content and then performs in parallel static and dynamic analysis to extract useful features that are afterward used to feed a SVM classifier. The static analysis performed by CUJO simply tokenize

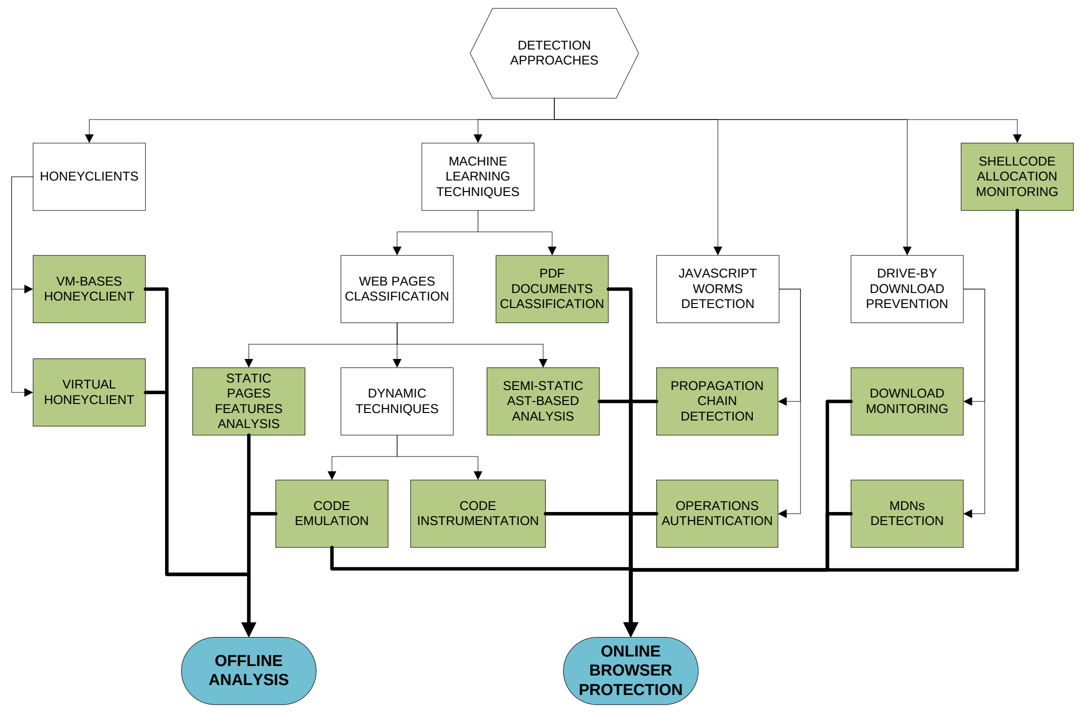
DETECTION APPROACHES
HONEYCLIENTS
MACHINE LEARNING TECHNIQUES
SHELLCODE ALLOCATION MONITORING
VM-BASES HONEYCLIENT
WEB PAGES CLASSIFICATION
PDF DOCUMENTS CLASSIFICATION
JAVASCRIPT WORMS DETECTION
DRIVE-BY DOWNLOAD PREVENTION
VIRTUAL HONEYCLIENT
STATIC PAGES FEATURES ANALYSIS
DYNAMIC TECHNIQUES
SEMI-STATIC AST-BASED ANALYSIS
PROPAGATION CHAIN DETECTION
DOWNLOAD MONITORING
CODE EMULATION
CODE INSTRUMENTATION
OPERATIONS AUTHENTICATION
MDNs DETECTION
OFFLINE ANALYSIS
ONLINE BROWSER PROTECTION

Figure 4.1: JavaScript malware detection approaches taxonomy and application fields.

the JavaScript code using a customized grammar, the dynamic analysis process instead leverages ADSandbox [27] to run the code in a virtual environment and record all the performed operations. The subsequent step taken by CUJO is to extract a set of *q-grams* (set of q terms interleaved with spaces) from each line of tokenized code and each operation record: these are the features used in classification. To finally obtain the classification outcome the set of features is used as vector space representation of the page: a value is given for the presence (1) or the absence (0) of a certain q-gram in the feature set and then used in the calculation of the detection function for the current page. Although CUJO may be deployed without browser modifications and the evaluation shows that it provides good detection performance, it adds a non negligible overhead to the page visualization process, that can reach the magnitude of seconds for JavaScript intensive pages like facebook.com: as previously said the lightness is a key point for the effectiveness of a malware detector.

ADSandbox, employed by CUJO to acquire data about the dynamic execution of JavaScript code, is a detection tool that rather then classify pages basing on a model, uses the extracted information to find patterns likely generated by the execution of malicious code. The entire execution is demanded to SpiderMonkey[1], the JavaScript engine included in Mozilla Firefox. This approach lead to a lower detection rate with respect to machine learning tools, simply because an attacker has full freedom to obscure the execution trace of his code thus avoiding detection, that is realized with a small set of manually generated regular expressions. Although this approach has possibly higher detection rate than simple signature matching systems (§ 2.2.3), they are not comparable with those of other tools described in this section.

Another machine learning approach is JSAND [3], a detection tool that use anomaly detection and emulation to identify malicious JavaScript code hosted in a web page. The main difference with CUJO is that instead of running the code in a JavaScript engine, JSAND sets up a complete execution environment that consists of the web browser framework HtmlUnit[2] and the JavaScript interpreter Rhino[3]. This architecture allows JSAND for a more comprehensive analysis because relying on a web browser (although emulated) makes possible to inspect more behavior due to the possibility of leverage, in addition to the complete JavaScript execution context, also the DOM API; this feature is very important because is always possible for an attacker to hide some code or data inside the HTML markup and retrieve it when needed. The use of HtmlUnit as browser emulation framework gives a chance to defeat certain obfuscation techniques based on environment fingerprinting (§ 3.3.1), this is possible by setting different browser personalities (in the form of different exposed APIs or environment variables value), possibly revealing variation in the behavior. Another feature that enlarges the analysis capabilities of JSAND is the possibility to emulate any browser external plug-in (e.g. ActiveX controls) and monitor the operations executed over it. During the analysis of the page the tools keeps track of many features that are used in classification: diversely from CUJO the set is fixed (although easily extensible) and manually defined: the learning process generates only the threshold values for each feature i.e. the model of a benign page. The features employed in classification (performed using a Naïve Bayesian classifier) are comprehensive of many types of aspects that can highlight the occurrence of an attack, namely: *redirection and cloaking*, *deobfuscation*, *environment preparation* and *exploitation*. Considering the entire set of functionalities exposed by JSAND is clear that it cannot be included in a web browser, as a matter of fact its analysis time requirement is in the order of tens of seconds per page, unacceptable for an online use. This enormous time required for the analysis is however not very surprising because

[1]SpiderMonkey, `https://developer.mozilla.org/it/docs/SpiderMonkey`
[2]HtmlUnit, `http://htmlunit.sourceforge.net`
[3]Rhino, `https://developer.mozilla.org/en-US/docs/Rhino`

JSAND is a tool built explicitly for offline analysis: its main usage can be as blacklist generator when coupled with a crawler. Its developers also built up a public available web-service (called Wepawet[4]) that analyzes samples submitted from the crowd and gives back a detailed analysis report; doing this is possible to acquire other samples of both benign an malicious pages to refine the anomaly detection model. To further improve the performance of the web service many other tools have been developed as described in [19],[4] and [13].

Previously described approaches were unsuitable for an effective implementation inside a web browser due to design specifics or high overhead introduced by the analysis. The following tools overtake this limitations being designed specifically for in-browser detection and adding a (quasi)negligible latency to the page visualization process.

ZOZZLE [9] is a detection tool that performs a mostly static analysis of the JavaScript code directly in the browser. The main difference with respect to other approaches is that the analysis is static, i.e. relies only on features extracted directly from the code, without executing it. More precisely the features are extracted from the abstract syntax tree (AST) representation of the code; they are then submitted to a Naïve Bayesian classifier that gives the final response. This type of analysis, that doesn't make use of any emulation technique to extract features, is faster than dynamic analysis but allows to reach comparable performance. Unfortunately, due to the adoption of obfuscation techniques, a complete static analysis is infeasible because the most of the malicious behavior became visible only at run time. To address this problem ZOZZLE is integrated with the browser JavaScript engine, making possible to hook the native functions that are commonly used in the deobfuscation process (e.g. `eval`, `unescape`, `setTimeout` etc.). With this device ZOZZLE is able to retrieve the unobfuscated version of the code, extract the features from it and send them to the classifier. Although the performance of ZOZZLE are quite high and comparable with similar tools, it suffers of several limitations: (i) it is weak respect to simple variable renaming or polymorphism and (ii) creating a new reference to a built-in function will completely bypass the hooks, making impossible to obtain the unobfuscated code . These limitations pose a great problem on the applicability of the approach because with simple modifications even existing malware can leave its malicious behavior undetected.

The last machine learning tool here described is ICESHIELD [8], a detection tool that uses code instrumentation and functions hooking to extract features from the page and block any exploit attempt. The type of analysis performed by ICESHIELD is similar to that of JSAND in the sense that a set of heuristics are kept to monitor the fitting

[4]Wepawet, `http://wepawet.iseclab.org/`

of the running code to the model of benign code derived during the training phase; in addition IceShield can dismantle the attack once detected, simply changing the suspects parameters values passed to the the vulnerable function. The instrumentation is performed over all the native methods of `window`, `window.document` and the DOM APIs, leveraging the feature `Object.defineProperty()` available since ES5[5]. This method allows to *freeze* objects properties simply setting the descriptor *configurable* to *false*, in this way any subsequent modification will be denied. One of the peculiar characteristics of IceShield is that it doesn't require any browser modification to be implemented because it is written entirely in JavaScript, it can hence be deployed on different devices such as desktop PCs, notebooks and smartphones, simply including it in the page or injecting it using a proxy; the only requirement for the effectiveness of the tool is that the browser and the JavaScript engine must implement correctly the ES5 standard. If the aforementioned condition holds the tool is also granted to be tamper resistant, so that even if the attacker is aware of its presence he cannot disable it. Anyway limitations subsist also when the specifications are followed: if the page loads another context (e.g. creating an `iframe`, doing a redirection etc.) the tool is completely bypassed because the browser natively makes different context isolated from one another. To avoid this evasion technique is possible to deploy IceShield as a browser extension instead of a library, in this way each page loaded in the browser will be subject to the analysis, however loosing the portability of the tool. As already said the principal requirement is the fitting to ES5 specification: this is a limitation that cannot be easily solved in older versions of browsers that are still widely used but that are not capable to provide such support and are not likely update.

Events handling. A desirable functionality, that has not been implemented in the previously described tools, is the possibility to trigger events in the user interface to execute their associated routines. Such a feature would be very useful because malware authors can leverage this lack in detection approaches to hide their malicious code that reveals itself only when a legitimate user interacts with the page. Although the automatic GUI stimulation is an already faced problem [14] it has to be addressed correctly taking into account that a repeated stimulation with different ordering among the events may be necessary to highlight the malicious behavior.

The need of dynamic analysis. Here are listed a number of observations that try to emphasize which are the most desirable characteristics that a malware detector should

[5]The ECMAScript Language Specification, `http://www.ecma-international.org/ecma-262/5.1/`

possess. The basic concept that underlies these observations is that static analysis is no more a viable solution as it can be easily bypassed with simple obfuscation techniques that are become a standard for today's malware.

- **Features type:** tools that employ dynamic features instead of syntactic features, like JSAND and IceShield, have better detection performance due the generality of the approach that can highlight intrinsic characteristics of many classes of attacks: they abstract from "how malware looks like" concentrating in "how the malware behaves". Even if an attacker can still try to build up a script that performs malicious actions showing a behavior similar to the model of benign code learned by the detector, this event is made less and less likely with stepwise refinements of the model.
- **Code coverage issue:** as previously said the main obstacles in which an approach based on dynamic analysis incurs are runtime overhead and reduced code coverage. Anyway, in the case of client-side detection, code coverage is not an issue because the goal is to detect attack attempts that are taking place and not discriminating between benign or malicious scripts. This concept must be taken into account when realizing a dynamic analysis detector for the client machine because the attention must be focused only in making the tool fast enough to not slow down the browser.
- **Browser modifications:** even if in the previous survey the need of browser modifications to implement a detector has been considered a disadvantage, due to difficulty to provide protection to legacy software, is obvious that integrating the detector inside the browser (or the JavaScript engine) allows to exploit synergies that possibly lead to more efficient, and hopefully more effective, detection. This should be a hint for future research efforts that making use of this approach can realize dynamic client-side detectors that effectively protect the users.

Malware data sets construction. Tools based on machine learning techniques are first trained with samples of JavaScript code (both benign and malicious) to build the model that will be used during the analysis phase: is obvious that build up such data sets correctly is crucial for the effectiveness of the final tool. The first choice for a researcher to acquire useful data for training and validation is the manual inspection of several web pages, but this strategy is hard, time-consuming and error prone. A different way of dealing with this problem, followed by the majority of researchers, is to rely on widely accepted lists of both benign and malicious web sites e.g. Alexa Top Sites[6] for benign

[6]Alexa Top Sites, `http://www.alexa.com/topsites`

pages and Google Safe Browsing API [7] for malicious ones.

4.2 Heap-spraying detection

Employing JavaScript as a vector to deliver malware to the victim user, an attacker can easily enhance his probability to have a successful exploit by using the well known technique named heap-spraying, widely adopted in traditional malware. This technique tries to increase the probability of a successful blind jump to a shellcode stored in memory by allocating multiple instances of the same shellcode. Doing this the attacker can likely reach one of the copies been instantiated, also overcoming the limitations posed by the address space layout randomization. Because of the knowledge of this habit in malware construction researchers have proposed solutions that address specifically this problem, developing tools that detect the attempts to set up an heap-spraying attack; these tools looks for shellcode-like strings in the heap and so can detect all types of attacks which require the allocation of executable code and not only heap-spraying attempts.

Egele et al. describe in [28] a technique to detect attempts of instantiating shellcode strings in the heap. To perform this task they modified the JavaScript engine Spider-Monkey adding code that analyzes each string being declared, checking if it contains executable x86 code. The check is performed leveraging the library *libemu* which looks for a valid x86 instructions sequence starting from each character of the string. Perform the check at declaration time allows to detect the presence of the shellcode before a memory vulnerability causes its execution, making the exploit fail. Unfortunately analyze every string introduces a great overhead to the page visualization process, that reaches the magnitude of seconds and doubles the time needed to show the page. To improve performance some optimization have been implemented like factorizing the analysis of related strings and check only strings that are directed to external components (like plug-ins); anyway even with these optimizations the total time remains 1,5 times higher than the native one. Even though the time requirement is quite limiting the approach is interesting because it makes (hopefully) impossible for an attacker to trigger vulnerability in the browser or in its external components.

A similar approach is implemented by NOZZLE [29] that diversely from the previous tool doesn't analyze the sole instantiated strings, but also all the other heap allocated objects; the object scanning is also asynchronous and periodical because, unlike strings, heap objects are mutable. To enable the object scanning the native routines that handle creation and deletion of objects in the heap have been instrumented adding code that

[7] Google Safe Browsing API, `https://developers.google.com/safe-browsing/`

keeps track of the objects currently allocated. The analysis, performed by NOZZLE using a pool of parallel scanning threads, interprets objects in the heap as x86 code sequences trying to extract a control flow graph (CFG). For each valid basic block in the CFG NOZZLE calculates a metric that is then combined with those of other objects to derive two global heap health metrics: the *attack surface area* and the *normalized attack surface area*. This metrics are the key points of the entire analysis because the decision to report an attack is based on their exceeding of two predefined threshold values. Looking at the evaluation results the overhead imposed by NOZZLE is quite high (in general from 150% to 200% of the native time) but can be heavily reduced introducing sampling over the object to be scanned: sampling can decrease the overhead by more than 50%, with a loss in accuracy lower than 10%, with a heavier sampling is possible to lower the overhead by 90% keeping the error lower than 20%.

4.3 JavaScript malware detection in PDF documents

A relatively new vector used to deliver JavaScript malware is the Printable Document Format (PDF), widely adopted to exchange documents and already leveraged to distribute traditional malware due to the plenty of well known vulnerabilities that plague most of the readers (including Adobe Reader). The PDF format has become very interesting also for JavaScript malware authors because it gives chances for an even more effective (and easy) obfuscation of the malicious code: as a matter of fact the PDF standard provides natively support for many types of encoding/compression routines that make difficult for a detector to analyze all the code; in addition, due to the complex structure of a PDF document, JavaScript code can be placed in a variety of locations, making difficult even to find the code inside the document. Together with all the above issues, JavaScript in PDF documents still preserve all the properties highlighted in the case of web pages: dynamic code generation, code loading from external sources etc.

The first tool that tries to detect malicious JavaScript code in PDF documents is MDSCAN [32]. The analysis strategy employed by MDSCAN consists in parsing the document thereby extracting all the defined objects which can virtually contain JavaScript code. This operation is not as straightforward as it seems because is possible to place objects without listing them in the global cross-reference table of the document, so that their extraction requires the entire parsing of the document; these object are still be visible due to the best-effort strategy employed by PDF viewers. Fundamental for the object retrieval process is also the Acrobat API emulation that allows to extract objects created by an attacker with this API and that can be accessed only through it.

Once all the objects have been collected, is necessary to extract the code from them (searching for the `/JS` key) and reorder related chunks to reconstruct the complete code. Knowing that the main objective of an attacker is to divert the normal execution of the PDF viewer, the detection approach is to run the code in an instrumented version of the JavaScript engine SpiderMonkey that scans all the allocated string objects using the shellcode detector Nemu, looking for attempts to set up an exploit. The approach is analogous to that described in [28] and also in this case the overhead imposed is quite high, anyway this is not a great limitation being MDScan a tool built to be a standalone PDF scanner, ready to be included in anti virus software, intrusion detection systems etc. The use of a partial emulated version of the Acrobat API MDScan cannot detect correctly some document that make use of any non-implemented method of the API, anyway this lack can be easily overcome with stepwise enrichment of the emulated API; this in turn can make arise problems of inconsistency between the original and the emulated version of the API, where a particular effect leveraged by the attacker is not properly mimed.

A detection approach similar to MDScan is PJScan [33] which share the same basic idea but performs the code extraction and the classification in a different way. Diversely from MDScan the extraction is demanded to a modified version of the open source PDF parser Poppler[8], which recursively scans the dictionaries defined in the document to find locations that can contain JavaScript code, once it identifies them it searches for the key `/JS` (required to make scripts executable), extracts the code samples and decode/decompress them when needed. With respect to the previous approach PJScan has a more efficient object retrieval phase due to a targeted object extraction guided by the dictionaries. After their collection code samples are executed inside SpiderMonkey which has been modified to create a tokenized version of the code: this stage and the subsequent classification process are quite similar to the ones performed in [7], being employed a SVM classifier and a set features extracted from tokenized code. The approach taken by PJScan is subject to a limitation that hampers its application in the real world, it is in fact unable to detect correctly documents where the malicious payload is retrieved from a location that, following the PDF specification, shouldn't contain JavaScript code; it is noteworthy that even the approach taken by MDScan doesn't solve completely this issues because it looks for legitimate JavaScript inclusions (denoted by the `/JS` key), while an attacker can easily embed it as text and evaluate it at run time (e.g. via `eval`).

The great limitation of the previous tools is the inability to provide a complete protection against malicious PDF, focusing only on attacks that involve JavaScript code,

[8]Poppler, `http://poppler.freedesktop.org/`

they do not cover threats coming from embedded executable, external redirections, malicious Flash objects etc., anyway many of these issues are addressed by other works [34, 35]; a desirable functionality would be to integrate all these approaches creating a stand-alone tool for an in depth analysis of PDF documents.

4.4 Support tools

To enhance detection capabilities and performance of analysis tools researchers have developed many strategies. Although some of them were created to extend the functionality of a specific analysis system (like Wepawet) their approach is general and can applied to other detection tools. These tools focus their attention in fighting evasion techniques employed by the attackers to leave undetected the malicious code, they are not supposed to be implemented at client-side because, in addition to requiring a dedicated infrastructure, do not add any additional layer of protection.

Revolver [13]. Often malware authors exploit obfuscation and fingerprinting techniques to evade detection without modifying their malicious code, this scenario gives the chance to defeat those evasion techniques by leveraging a previous detection of the same code. REVOLVER is a tool built for the detection of evasive JavaScript malware, based on the idea that code that is similar should have the same classification outcome: different results for code samples considered similar can be symptom of an evasion attempt. The similarity check performed by REVOLVER is based on the analysis of the AST representation of the script, that makes possible to ignore superficial modifications (e.g. variable renaming) and at the same time allows an easier identification of the evasive code: the tool is indeed capable to identify the code that introduces control flow changes in the script (maybe for implementing fingerprinting). To perform its analysis REVOLVER relies on a dataset of scripts classified by an oracle for which the AST has been extracted and compared for similarity identification; pairs of similar scripts with different oracle outcomes are then divided in four categories on the base of differences found in the AST, trying to identify the evasion technique applied: the categories are *evasions* (fingerprinting), *code injection* (insertion of malicious code in benign scripts), *data dependency* (use of packers) and *evolution* (general malware modification). The limitations in which REVOLVER incurs depends on the differences between the oracle and the regular browsers which can be exploited by an attacker to leave undetected the malicious behavior: the evaluation of REVOLVER, performed using Wepawet as oracle, revealed a number of possible evasion techniques that remain hidden, anyway other issues can be identified using different oracles whenever they don't mimic exactly a real browser environment.

Kudzu [14]. Find client-side code injection vulnerabilities in web browsers is the purpose of the tool KUDZU, which inspects the execution space of a script using symbolic execution. Symbolic execution consists in running the code maintaining for each variable, instead of concrete values, symbolic formulas which define the values as a function of the inputs: during the execution each operation assigns to the result variable the formula obtained combining the operand formulas properly. Keeping track of the encountered conditionals, and their outcome, KUDZU reconstructs the condition that allowed to reach a certain execution point; afterward this condition is passed to a constraint solver that creates new input values to explore a different execution path. Given the possibility to include code associated to events, KUDZU employs a GUI exploration component that has the purpose to sequentially trigger every registered event. To identify the presence of a code injection vulnerability KUDZU checks whether untrusted data (e.g. URL parameters, data from HTTP channels and also user input) flows to critical sinks (e.g. `document.write`, `eval` etc.) and in that case analyzes the code to verify the sanitization of the untrusted data; the constraint solver is employed also to verify the sanitization: if it finds an input that after the sanitization process still preserves certain characteristics defined for the specific sink, the tool marks it as an attack. To further verify the effectiveness of the alleged attack the input is fed back to the application and, if the expected behavior is detected, KUDZU reports an alarm.

Rozzle [12]. The adoption of fingerprinting techniques poses non-trivial limitations to the analysis of JavaScript code due to the reduction of the code coverage, possibly leaving malicious behavior undetected. A viable solution to this problem is symbolic execution, already employed in KUDZU, a choice that has the drawback of a colossal analysis duration; a different approach would be to simulate different system configuration using many (possibly virtual) machines, with the hope to find the one that reveals the malware, but even this technique is not decisive because has huge hardware requirements and at the same time has no guarantees to find any malicious behavior. ROZZLE is a multiexecution virtual machine developed specifically to defeat fingerprinting techniques: it doesn't perform the analysis itself but amplifies the detection capabilities of a malware detector. The idea behind this tool is to execute simultaneously both branches in the case of environment dependent conditionals; the observation behind this strategy is that benign code is not likely to have environment specific behavior, that is instead common in malicious code that uses fingerprinting. To implement multiexecution environment dependent values are treated as symbolic (i.e. holding many values depending on conditions) and, similarly to what has been done in KUDZU, applying operands and conditionals symbolic values are propagated properly, choosing the correct one in each scenario. Although many tests show the effectiveness of ROZZLE in revealing more behavior during the analysis of JavaScript code, an attacker can still try to hide the malicious code: an example is the use of server side fingerprinting where after the environment detection a server provides different scripts on the base of system characteristics.

EvilSeed [4]. EVILSEED is a tool built for enhance the effectiveness of a crawler which has to provide URLs to a detector. Standard crawlers visit web pages blindly, simply following the outgoing links placed inside the current page, this produce a result that have no guarantees to contain any malicious URL. To solve this problem EVILSEED smartly leverages the crawling capabilities of the search engines to retrieve a set of URLs that is likely to contain many malicious pages. The process starts from a defined set of malicious URLs which are processed to extract information that can be used to perform query over search engines. This task is performed by a set of specialized components (called gadgets) that create different types of queries to exploit many search engine features. After retrieving responses from the search engines EVILSEED employs an oracle to fast discriminate between benign and malicious URLs, adding the latters to the initial set for another iteration of the algorithm. Given the features used by EVILSEED an attacker can still try to leave out certain pages from the search engine response, at the cost of a reduced page visibility in the search results and a reduced probability of attracting victims. The only requirement for a useful usage of EVILSEED is the construction of a solid initial set of malicious URLs which has to include the largest possible number malicious page classes, in order not to leave out entire portions of malicious URLs. EVILSEED can be effective in adding a further protection layer when users use search engines: implementing the tool directly in a search engine infrastructure allows for a more effective an efficient creation of blacklists that are then used to filter out undesirable entries from the search results.

Prophiler [19]. Given the analysis time requirements of an offline dynamic detection system is obvious that is undesirable to waste time and resources analyzing pages that are very unlikely to be malicious. It would be useful to have a filter that discards benign pages, sending to the analysis system only those which are suspected to contain malware; the fundamental requirement of this filter is that it must be times faster than the analysis system, otherwise its benefits are reduced. Knowing that the high time requirements of analysis systems are due to the use of dynamic analysis techniques, is clear that the filter cannot exploit such strategies but should rely only on static analysis to perform its job. Prophiler implements exactly this approach, quickly discarding benign pages while sending the others to a detector. The fast classification implemented by Prophiler relies on a set of static features extracted from the different sections of the page, namely: *HTML markup*, *JavaScript code* and *URL*: these features are quite general and cover different aspects of many classes of attacks, making difficult for an attacker to evade detection, but even if the evasion has success a small fraction of supposed benign pages are sent to the analyzer, making possible to have an estimate of the false negative rate. As said before the purpose of Prophiler is to prevent the analysis of likely benign web pages thus saving resources, but is important to notice that a false negative in the classification can leave undetected a true malicious page: this issue has to be taken into account when developing the model so that it can be realized accepting more false positives with respect to false negative, as they cause only a reduction in benefits. The evaluation of Prophiler has been made together with the analysis system Wepawet, showing it effectiveness with reduction in the number of analyzed page by 87%, saving about 400 days of processing. Similarly to EvilSeed this tool is eligible to be added in search engines to further harden their blacklisting systems.

EarlyBird [36]. Looking at the previously described detection systems, none of them face the problem of the early detection of an attack: all of them classify the page as benign or malicious only at the end of the execution, leading to potential harm to the victim. EarlyBird is a learning method optimized to limit the fraction of executed code needed to correctly classify a web page. The learning approach considered in EarlyBird is the one used in CUJO (which implements EarlyBird), but can be easily adapted to other machine learning models different from linear SVMs. The basic idea is to consider more important features related to malicious code that appear early in the execution by using two different weighting schemata: one (flat) for benign features an the other (which decreases with time) for the malicious one. Experimental evaluations show that using EarlyBird is possible to obtain a detection performance comparable to the one of standard detection tools by reducing the executed code by a factor of 2; anyway the methodology suffers of a non negligible vulnerability: if an attacker adds sufficient benign operations before the malicious code the tool may classify the page as benign.

Obfuscated code simplification [20]. Often security researchers need to manually inspect scripts suspected of being malicious to further confirm the decision made by a detector or to extract information about the attacks; an obstacle to this important task is the use of obfuscation techniques that make the job very hard to be done by hand. A viable solution would be to use a deobfuscator like jsunpack[9], but even a plain text version of the code can be still too complex. Lu and Debray described a technique to obtain a simplified version of a script starting from its original obfuscated version while preserving the core logic of the code: the produced script is equivalent to the original one considering its interactions with the JavaScript environment. The idea is to collect an execution trace of the selected script using an instrumented version of a web browser and then analyze the trace using the technique named *dynamic slicing:* dependencies between variables that affects the parameters of the system functions are tracked to highlight the core logic of the script; the final step is to convert the intermediate representation produced by the dynamic slicing to obtain the simplified version of the code. This technique can be widely adopted to deobfuscate most of the JavaScript malware samples found in the wild, anyway is possible for an attacker aware of the analysis attempt, to hamper also the simplified script inspection by adding unnecessary system calls to the core one: due to the inability of dynamic slicing to distinguish between relevant and irrelevant native function calls the produced script contains also the fake core logic.

4.5 Defenses against JavaScript worms

With the rise of AJAX-based Web 2.0 applications, especially with the birth of social networks, malware authors have developed a new way to distribute malware to victims: the so-called JavaScript worms. The name derives from the well known category of malicious software whose peculiar characteristics are to be stand alone and be able to self spread to infect other hosts (usually placing itself as e-mail attachment). A JavaScript worm exploit cross-site scripting vulnerabilities of a website to replicate itself: when a user visit an infected web page the worm is downloaded, executed inside the JavaScript environment and, together with its malicious actions, it tries to copy itself on the web server, usually infecting the user's profile in the case of social networks. The malicious actions performed by a JavaScript worm are analogous to those of any JavaScript malware, so the difference in the approach to mitigate this threat is to block the malware spreading after the infection of the host has already taken place. Given the enabling scenario that allows the propagation of JavaScript worms is clear that every

[9]jsunpack - a generic JavaScript unpacker, `http://jsunpack.jeek.org/`

tool built to counter them is no more than a stop gap measure: the only way to finally eradicate the problem is to fix the XSS vulnerabilities of the web application.

SPECTATOR [26] is a system that detects and blocks attempts of a JavaScript worm to replicate itself over a web application by tracking the chains of propagation of the same content: the tool detects a worm whenever it finds a unusually long propagation chain. SPECTATOR works in a web proxy and is almost transparent to the host, requiring only the possibility to save a cookie in the client browser: no browser modification are needed, and a few information must be stored at server side. SPECTATOR works by tainting pages at server side adding a tag to every uploaded content. For each HTTP request the proxy remove tags from the page that is then sent to the user together with a session ID, saved as a cookie; the extracted tags are associated to the session ID in the proxy and added to the propagation chain if the client supplies an HTTP upload request. To avoid wrong chain creation in the proxy a small JavaScript code fragment is added to the page allowing to inform the proxy to invalidate the current session when needed. Is clear that the effectiveness of this method relies on the existence of the session ID on the client machine, but even if an attacker tries to eliminate it, the proxy can block the propagation of the worm disallowing uploads without a valid session ID.

Another approach to counter JavaScript worms propagation is PATHCUTTER [25], which works at server side blocking unauthorized requests issued by the malicious code. The protection mechanism implemented by PATHCUTTER consists in dividing the web pages into isolated components, named *views*, associated with certain capabilities that allows them to execute only the legitimate operations that they are supposed to do: e.g. allow only the post form of Facebook to actually publish contents on a page while making impossible for a script, maybe injected in the page as a post, to do the same operation. The view separation is realized embedding each view in an `iframe` which refers to a pseudo-domain, obtaining native separation through the same-origin policy (§ 2.2.1): by doing this also an attacker aware of the presence of PATHCUTTER cannot interfere with its protection mechanism. The blocking of unauthorized operations can be performed at server side in two different ways: (i) embedding a secret token associated to the operation in the view or (ii) using a referrer-based validation that allows the server to verify if the view has the required capabilities . Both of these authentication methods cannot be tampered by malicious JavaScript code because the view separation makes impossible to steal the secret token, while the browser disallows a manual modification of the referrer field. Although the protection mechanism provided by PATHCUTTER can be applied without browser modifications, it doesn't come for free, in fact it requires a modification at server side to implement view separation and requests authentication:

the effectiveness of the tool resides in the correct implementation of these operations, analyzing all the possible vulnerable operations and the possible use cases.

Although SPECTATOR is effective in detecting JavaScript worm whatever the worm looks like and independently from the speed of spreading, it has the drawback of requiring the creation of a enough long propagation chain: this imply that a great number of hosts have to be infected and possibly compromised before the worm can be detected. For this reason an approach like PATHCUTTER is preferable.

4.6 Drive-by download prevention

In literature can be found tools that aim to protect the final user from the threats coming from the novel attack methodology known as drive-by download where malicious executables are downloaded and launched as result of a successful exploit of a browser vulnerability. These targeted approaches try to block the download of the malware executable when the vulnerability in the browser has already been exploited so they cannot protect from the damages coming from the execution of malicious JavaScript code.

BLADE [37] is a protection mechanism that blocks the execution of malware executables isolating them in a non-executable zone. The approach implemented by BLADE is based on the recognition of the user interaction needed to allow the download of a file: legitimate downloads trigger the creation of a confirmation window that allows the user to give the permission to start the process, differently file downloaded as result of a successful exploit of the browser does not present such a window. To distinguish between legitimate and illegal downloads BLADE is integrated with the system at kernel level, making possible to intercept screen modifications and hardware events representing the consent of the user to download the file. BLADE is composed by five modules (i) a *screen parser* that intercepts screen modifications to find donwload confirmation windows, (ii) a *supervisor* that, triggered by the screen parser, manages the detection process, (iii) an *hardware event tracer* that intercepts events related to actions on the confirmation window, (iv) a *correlator* that verifies the authorization for each initiated download and (v) an *I/O redirector* that isolates unauthorized files in the non-executable zone . Evading such a protection mechanism in a non-compromised machine is very difficult for an attacker because there are few chances to interfere with the kernel protection provided by BLADE from inside the web browser; as a matter of fact the evaluation show that BLADE identifies correctly all the unauthorized downloads with no false-positives nor false-negatives. Is it noteworthy that the protection provided by BLADE is completely agnostic with respect to the nature of the downloaded file, being it benign or malicious, in fact it doesn't protect the system from malicious executable downloaded by luring the user to confirm the download, in this case is useful to rely on common antivirus software.

An additional level of protection against drive-by download is provided by ARROW [38], a system for the automatic generation of signatures for URLs belonging to the so-called *malware distribution networks* (MDNs). A MDN is a distributed infrastructure built to manage efficiently the distribution of malware used for drive-by download infections, it is usually composed by many different servers widely distributed that allow to achieve both reliability of attacks and robustness to detection; the standard topology comprises several *redirection servers*, that balance the requests for performance purposes, and a limited number of *central servers* that deliver the malware. The objective of ARROW is to detect the URL of these central servers in order to compile blacklists that can be used to filter dangerous download requests. The detection performed by ARROW starts from HTTP traces retrieved during the analysis performed by a cluster of high-interaction honeyclients, which store also the SHA-1 hash of each downloaded executable. The identification of the MDNs is performed grouping together URLs that caused the downloading of files with the same hash, subsequently for each MDN the central servers are identified as the URLs that are shared by the majority of the HTTP traces; as a refinement step MDNs with shared central servers are merged together, keeping only the shared central servers. Once the MDNs have been identified ARROW starts generating the signatures in the form of regular expression: to this scope a tree structure is realized using the tokens extracted from the central server URLs, the signatures are defined on the base of the branches with the highest coverage. The lists of signatures produced by ARROW can be used both for network level protection (in proxies or search engines) and for browser filtering. Relying on honeyclient to acquire HTTP traces ARROW is subject to the intrinsic limitations that plagues these tools as described in section §4.7.

In the field of the defense against drive-by download attacks the previous tools provide sufficient coverage to protect from the most of the attacks now analyzed, they however are only a further level of protection that cannot substitute other detection mechanisms like those described in sections § 4.1, § 4.2 and § 4.5.

4.7 Honeyclients

The use of honeypots is widely adopted by security researchers to detect attacks that try to take control of a victim machine, allowing an analyst to monitor the operations performed by the attacker and collect information to harden the defenses. This type of infrastructure is built explicitly to be compromised and so is completely passive, conversely to detect attempts of drive-by download attacks (or study JavaScript malware in general) the machine must be active and capable to browse the web to reach malicious pages. This difference is highlighted using the term server honeypots for the passive one and client honeypots (or honeyclient) for the active one. As for their passive counterparts, honeyclient are much more useful as better they are in mimic the behavior of a real system: this leads to the distinction between high interaction honeyclients, which replicate almost all the aspects of the original system, and low interaction honeyclients, which instead provide only a limited set of functionalities. In addition is possible to differentiate between real honeyclients, which make use of a real browser, and virtual honeyclients, which emulate the browser environment in software. The use of a virtual instead of a real honeyclient allows to create an analysis infrastructure that can scale well, making possible to analyze many web sites in parallel; the drawback is that the develop of such tool requires great attention to exactly replicate the browser behavior, tricking a malicious web site in believing that it is interacting with a real browser. The other choice is to employ a pool of virtual machines which have an easier set up at the cost of higher hardware requirements. Even if honeyclients are widely employed to study JavaScript malware they suffer of an intrinsic limitation: they can emulate a single system configuration at time, this makes necessary the use of several honeyclients to defeat fingerprinting techniques. An attacker aware of the possibility to be detected by an honeypot can even try to detect it an set up targeted evasion techniques, as described in section §3.3.3.

The Strider HoneyMonkey system [39] is an analysis infrastructure that automatically detects JavaScript malware that exploits vulnerabilities in the web browser. The system is composed of a three-staged pipeline of VM-based high-interaction honeyclients in which, in each of the three stages, web pages are visited in different conditions; exploits are identified monitoring the creation of files or registry entries outside the scope of the browser. In the first stage multiple page are visited inside a single unpatched VM, so that old exploits can be detected; if an exploit is reported each single URL passes to the second stage and is visited in a dedicated unpatched VM that, in addition to the exploits detection, stimulates the user interface using monkey programs and keeps track of the redirections, sending the landing URLs to the system input to be analyzed: this redirection analysis is useful to identify malware providers that use other sites as front-end to attract victims. Finally, in the third stage, URLs coming from the second stage are visited inside a dedicated, fully patched VM, in order to detect zero-day exploits;

URLs reaching the last stage are kept for repeated analysis to detect future zero-day exploits. An approach like Strider HoneyMonkey provides useful information about drive-by download attacks and attacks trying to take control of the victim machine in general, but lacks in detecting those attacks that remains in the scope of the web browser like cross-site request forgery: this limitation is not easily solvable because requires the instrumentation of the web browser.

An example of virtual, low-interaction honeyclient is PhoneyC [40] which emulates a real browser to provide information about possible attacks coming from a web page. The architecture of PhoneyC consists of a web crawler and an analysis engine, these components succeed in both interact with web sites as if they were a real browser and revealing JavaScript code behavior despite encryption and obfuscation techniques. The crawler and the analysis engine can act with different browser personalities that possibly reveal hidden code behavior during the execution inside the JavaScript engine SpiderMonkey. In addition, with an overriding of the native method `eval`, is possible to acquire the unencrypted version of the code. The analysis engine implements a detection mechanism based on the observation of known vulnerabilities in the current browser configuration (e.g. native methods, ActiveX objects, browser extensions etc.): this is done by creating JavaScript objects that expose the core functionality of the vulnerable methods while checking, and if necessary sending an alert, if parameters value are parts of known exploits. The use of vulnerabilities modules enhance the detection capabilities of the tool with respect to other approaches that need the external components installed in the system, in contrast this methodology requires the manual creation of such objects: the limits posed by this approach depend on the number of modules implemented. The limitations of PhoneyC lies in the incomplete emulation of a real browser: for example when the tool impersonate Internet Explorer, an attacker can easily identify SpiderMonkey thank to its differences between the native JavaScript engine.

Chapter 5

Securing JavaScript

Modern web pages host services exposed by different providers to enhance their user experience: examples of such services are the library JQuery[1], which eases the development of web pages, the map service provided by Google[2] and also contextual web advertisements. Such pages (called *mash-ups*) integrate untrusted third-party services including remote scripts that are retrieved by the browser and executed whenever a user accesses the page. Unfortunately, due to the inclusion paradigm dictated by the JavaScript standard, a number of security threats arise: the main problem is that every remote script included in a web page has the same capabilities of a native script and so malicious code has full freedom to interact with the page content and access its sensitive information. The issues arising from the inclusion of malicious scripts are particularly worrying in the case of web advertisement because, due to the wide presence of ad networks, is not always clear where the included content comes from. In the case of remote scripts inclusion the same-origin policy is useless because the real source of the script is not considered as its origin, which instead is the same of the page. A simple solution to confine untrusted scripts is to include them in an `iframe`, in this case the complete separation is done through the same-origin policy, anyway even when a minimal form of communication between host page and script is needed, this approach is not handy because an ad-hoc channel based on `PostMessage` must be manually developed. In [41] the problem of third-party content inclusion is faced analyzing how it is performed in practice: the results are quite scary because they show that also popular web sites trust JavaScript providers that can be easily compromised by an attacker, together with highlighting common programming practices that further aggravate the problem. Be-

[1]jQuery, `http://jquery.com/`
[2]Google Maps, `https://maps.google.com/`

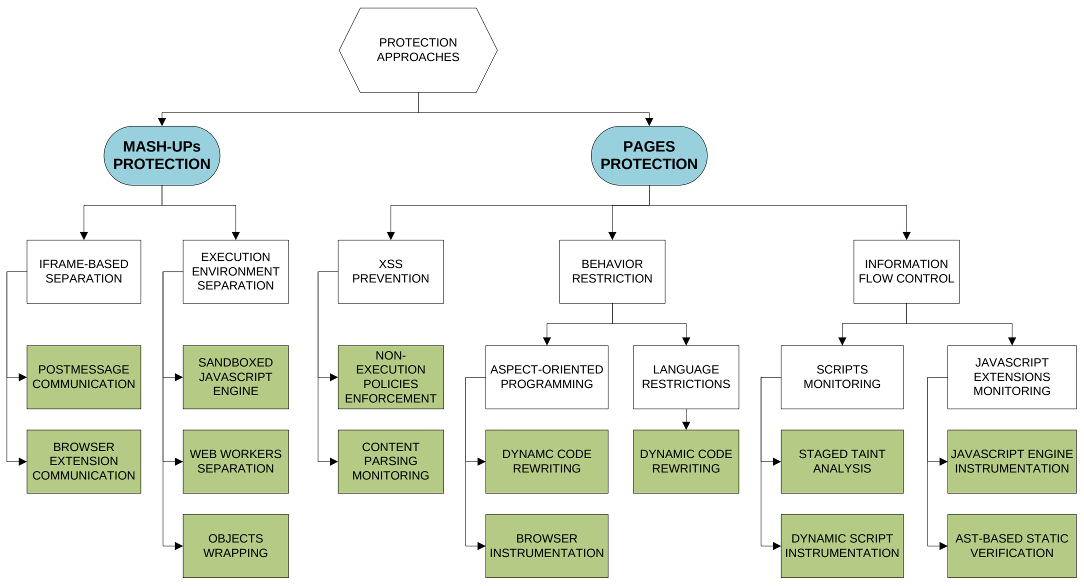
PROTECTION APPROACHES
MASH-UPs PROTECTION
PAGES PROTECTION
IFRAME-BASED SEPARATION
EXECUTION ENVIRONMENT SEPARATION
XSS PREVENTION
BEHAVIOR RESTRICTION
INFORMATION FLOW CONTROL
POSTMESSAGE COMMUNICATION
BROWSER EXTENSION COMMUNICATION
SANDBOXED JAVASCRIPT ENGINE
WEB WORKERS SEPARATION
OBJECTS WRAPPING
NON-EXECUTION POLICIES ENFORCEMENT
CONTENT PARSING MONITORING
ASPECT-ORIENTED PROGRAMMING
LANGUAGE RESTRICTIONS
DYNAMC CODE REWRITING
BROWSER INSTRUMENTATION
DYNAMIC CODE REWRITING
SCRIPTS MONITORING
JAVASCRIPT EXTENSIONS MONITORING
STAGED TAINT ANALYSIS
DYNAMIC SCRIPT INSTRUMENTATION
JAVASCRIPT ENGINE INSTRUMENTATION
AST-BASED STATIC VERIFICATION

Figure 5.1: JavaScript protection approaches taxonomy.

side the inclusion of untrusted scripts in web pages, security threats related to the safe execution of JavaScript code come also from evil pages or benign pages that have been compromised. In this chapter are reported techniques developed to make the execution of JavaScript code more secure, both in case of native and third-party imported scripts. In figure 5.1 is reported the global taxonomy of the protection approaches described in this chapter, organizing them by objective and then by technique.

5.1 Sandboxing frameworks

A sandboxing framework is a JavaScript library that eases the developer in confining untrusted content in a web page, so that it can execute without risks for the browser. Using such a framework is possible for the developer to create ad-hoc constructs that autonomously provide isolation, requiring only the definition of policies specifying any form of interaction needed to let the mash-up operate properly. This general approach is typical of this category of tools and requires only knowledge of the interactions between the imported scripts and the native page (often called the *integrator*) to be implemented, most of the times without the need of heavy re-factoring of the latter; the drawback is that a wrong configuration of the policies can hamper the entire sandboxing procedure. The differences in the approach taken by the following tools lie in the way they implement isolation between trusted and untrusted content and how the support for the necessary interactions is realized.

A first example of sandboxing framework is AdJail [42] which implements the `iframe` isolation strategy previously described with automatic management of inter-frame communication. The idea behind AdJail is to run untrusted scripts in a replicated environment (an `iframe`, called *shadow page*) which contains only the parts of the original page with which they can interact, every action performed in each environment is sent to the other making possible to (i) replicate the allowed untrusted operation in the original page and (ii) keep the shadow page informed of events that happen . The protection of the the integrator is ensured by applying the policies to the operations performed by the untrusted script, deciding whether they must be replicated in the original environment. The policies enforced by AdJail can allow the untrusted script to access HTML elements in writing and/or reading mode and express also the possibility to employ other technologies such as images or Flash objects. For a developer is easy to embed policies in a existing web page because they are expressed as a couple of `<permission:value>` inside the attribute `policy` for each HTML tag, every not specified permission implicitly assumes the most restrictive value. The communication infrastructure is realized inserting in both the pages a tunnel script that exchange data to its counterpart via `PostMessage`. AdJail is an effective protection mechanism built to secure advertisement inclusion in web pages, it allows the correct functionality of the ad network that collects information about the visualizations of the advertisement for billing purposes; the drawback is that the approach assumes a limited communication between the integrator and the untrusted content, making difficult to handle high interaction with the original page: example of such limitations are the impossibility to give access to certain objects defined in the original web page and restrict object usage once a reference has been granted.

Another sandboxing framework based on inter-frame communication is Mashic [43], a compiler which aims to realize automatic generation of secure JavaScript-based mashups starting from existing code. The approach implemented by Mashic (called PostMash) is described by Barth et al. in [44] and involves `iframe`-based separation together with inter-frame communication via the `PostMessage` API: in both the integrator and the embedded `iframe` are placed libraries that provide the communication infrastructure and in addition the library placed in the `iframe` exposes a public interface of methods accessible by the integrator. Unlike in AdJail is possible for the integrator to establish tight dependency with the hosted script having the possibility to invoke its methods passing as parameters both values and entire JavaScript objects; is also impossible to define policies for the execution of the hosted script. Using MASHIC is possible to automatically run the hosted script in a controlled environment where it can access only the properties which the developer allowed it to obtain. The limitation of this approach lies in the possibility of a sole one-way communication from the integrator to the untrusted script: this doesn't make MASHIC useless, but restricts it applicability only to those case where the one-way communication is sufficient for the correct operation of the mash-up (e.g. the embedding of Google Maps or a YouTube video).

An approach that slightly deviates from the previous is OMASH [45], a sandboxing framework that employs `iframe`-based isolation but tries to reduce the runtime overhead due to the serialization providing a communication infrastructure based on a browser extension. In OMASH each untrusted content, embedded in a `iframe`, exposes its public interface through the definition of the function `getPublicInterface` that is public available for other parties, the only limitation is that the value passed as function parameter must be basic types (like `string` or `number`). The possibility to share a public interface with other frames and the ability to call methods belonging to the interface are the core of the entire method and are made possible through a modification in the CAPS system of Mozilla Firefox; such a modification translates in practice in an exception at the standard same-origin policy enforced by the browser, that is still preserved for normal behaviors.

Most of the limitations that plague the previous approaches derive from their choice of implement isolation using `iframes`, requiring ad-hoc strategies to overcome the same-origin policy when needed, in addition the intense use of `PostMessage` API adds a non negligible execution overhead due to the serialization/de-serialization process required to send complex objects as strings. Another important concern is that `iframe`-based isolation only protects the integrator from being corrupted but doesn't provide any type of protection to the browser. Realize isolation inside the same frame, even requiring more accurate implementation to effectively secure a mash-up, allows to support deep interactions between the integrator and the untrusted scripts, giving the possibility to exert a fine-grained control over third-party script capabilities: realizing a fine grained control over the actions performed by the imported scripts is possible to ensure also the security of the browser blocking suspicious operations.

JIGSAW [46] is an isolation mechanism that secures mash-ups allowing the embedding of third-party scripts inside the main frame, adding to the JavaScript syntax the possibility to define properties visibility using the common concept of public/private fields. Each untrusted script is loaded in a JIGSAW container (called *box*) which allows its execution while blocking all interactions with other entities included in the page: this concept is similar to that of `iframe` but adds further restriction: (i) it limits intra-origins communications by default and (ii) blocks network accesses for nested inclusions . This `iframe`-like isolation is extended by the possibility of specifying policies that grant/deny functionalities to the included script and allows for synchronous function calls, making possible to invoke methods belonging to other boxes. For each top-level box JIGSAW provides many resources such as dedicated DOM tree, storage space, network access and a visual region, allowing the correct operation of the script; any nested content included by default has no access to parent resources, but the parent can decide to share them. Whenever an object leaves its original box, both as parameter or function result, it is encapsulated inside a so-called *surrogate object* which enforces the visibility constraints defined for the object itself. When a property has been set as private it cannot be read, wrote or deleted outside its originating box: the JIGSAW approach is conservative because when the visibility modifier for a property is not specified it is assumed to be private. The functioning of JIGSAW is based on a compiler that translates the code with modifiers to JavaScript with runtime security checks that enforce the constraints defined with the visibility modifiers, in addition a JavaScript library is needed to provide constructs such as box and surrogate objects creation. The only limitation of JIGSAW addresses the usage of prototype objects for surrogate and built-in objects, but the great majority of benign code doesn't exploit such feature and so won't be affected by the limitation.

ADSENTRY [47] is an ad isolation mechanism built to allow the secure inclusion of web advertisement, mediating the interaction with the original page according to customizable policies. The approach implemented by ADSENTRY is based on the execution of all untrusted code in a so-called *shadow JavaScript engine* (i.e. a JavaScript engine that runs in a sandbox) that prevents any access to the integrator, in addition interposition for the DOM access allows to carry out the operations on the original DOM tree, according to the defined policies. The approach is similar to ADJAIL in the way the access to the real page is provided, anyway ADSENTRY, in addition to a different architecture that requires browser modifications, provides additional features like the possibility to have tight dependency with the included ad and also the possibility to protect the browser from malicious scripts that try to exploit vulnerabilities.

An extreme approach to JavaScript sandboxing is JS.JS [48] which adds the entire JavaScript interpreter SpiderMokey (translated in JavaScript) on the top of the native interpreter included in the web browser, having so complete control on the execution environment in which the untrusted script runs. The sandboxing is realized removing by default all references to global objects and native functions from the sandboxed environment while allowing the developer to provide such references in a controlled way, specifying even type constraints for the methods parameters; by doing this the included script can behave only as the integrator designer allowed it to do. Every operation performed by the script on its virtual DOM is intercepted by a component named *mediator*, which can accept or reject requests and, if needed, reflects modifications on the real DOM. Even if the approach provides isolation effectively, requiring only the inclusion of a library ad no browser modification, it has non negligible drawbacks which not recommend its application: (i) the need of manually provide required references to the sandoboxed scripts is a tedious task especially when the number of included scripts or references is high and (ii) the interpretation performed in JavaScript causes a heavy slowdown of the included scripts execution of two orders of magnitude (on average) with respect to normal execution .

A different architecture that implements scripts isolation and DOM interposition is TreeHouse [49], which realizes separation using the novel features of the *Web Workers* provided by the current state-of-the-art web browser. Web Workers are the counterpart of threads in the context of web browsers and have been introduced with the HTML5 specification[3] to give the possibility of executing JavaScript code in a concurrent fashion. An interesting feature of these tools is that they provide a separate execution environment for each script, creating a native isolation mechanism that can be exploited to provide security in mash-ups. By default every access to the DOM is denied and the sole way to communicate is the message passing facility of `PostMessage`. In TreeHouse each Web Worker is equipped with a script (called *broker*) that sets up the environment to implement browser API interposition which allow a controlled communication between the Web Worker and the real DOM tree. To make instrumentation tamper-resistant is applied the freezing technique already used in IceShield (§ 4.1). The counterpart of the broker is the *monitor* component which, residing in the main page, sends events to the Web Workers and receives the operation requests that are applied according to the page author policies. The isolation mechanism realized by TreeHouse is easy to deploy, requiring no browser modification and limited page re-factoring, it is also more robust with respect to other approaches that require different origins to separate untrusted content and do not protect the application liveness from error occurring in the included scripts. With respect to js.js, the use of TreeHouse provides the same level of isolation, avoiding the enormous overhead added to the third-party script execution, causing only a limited slowdown in the operations that affect the DOM due to the usage of the `PostMessage` API.

[3]The HTML5 specification, `http://www.w3.org/html/wg/drafts/html/master/`

JSand [50] is a sandboxing framework which provides complete mediation to the execution of third-party scripts in a web page, allowing the secure inclusion of complex JavaScript frameworks. The approach is based on the use of an object-capability environment that limits the references that an untrusted script may access, leaving out security-sensitive resources. The enforcement is realized giving to a script only a limited set of object references and implementing the so-called *membrane pattern*, i.e. placing wrappers around sensitive objects to grant access with respect to policies defined by the designer; these wrappers, realized leveraging the Harmony Proxy API, can block operations performed on the wrapped objects and deny any direct reference to it from the untrusted script. The problem of making the integrator able to communicate with the sandboxed script is solved by the creations of the functions `innerEval()` and `innerLoadScript()` which evaluates new code inside the sandbox and loads a new script from a specified source respectively. The exposition of these functions makes possible to easily implement JSand in existing web pages requiring only a minor refactoring process that allows to preserve actual code, in addition JSand supports the entire JavaScript syntax and is backward compatible. The policy language accepted by JSand is more expressive than those used by other frameworks as it allows to define, together with simple stateless policies, also stateful policies that regulates the access capabilities of code in a sandbox depending on its past actions: e.g. access to XHR can be restricted if the script read from cookies database.

Considering the the actual trend in developing of web-based applications that moves from sole advertisements inclusion to complex JavaScript-based framework integration, the choice to safely include such components cannot be based on `iframe`s because it introduces an unacceptable overhead and too many obstacles to communication and development; TreeHouse partially solves those problems, but realizing inter-script communication via `PostMessage` keeps a considerable slowdown. Even if an approach like JSand is probably the right choice to provide the needed isolation with an acceptable overhead, a similar isolation mechanism provided directly by the browser can lead to the same level of security with still lower overhead; this type of approach is described in section §5.2.

5.2 Reference monitors

Reference monitors are tools built to enforce security policies during the execution of JavaScript code, depending on the way they realize enforcement policies different levels of granularity can be defined. The effect of a reference monitor is to block undesired behaviors and/or mediate access to sensitive resources. These tools are in a way similar to sandboxing frameworks described in section §5.1 but provide a wider protection applying enforcement to the whole JavaScript code and being active without additional developer effort. The deployment of a reference monitor can be realized both instrumenting the browser to be able to read and enforce policies, or via JavaScript including an ad-hoc library that sets up the enforcement environment.

Phung et al. described in [53] an inlined reference monitor (i.e. realized in JavaScript) that modifies the execution environment of a page through a source-to-source transformation of the code. The main objective of this reference monitor is to mediate access to security relevant objects and methods performing an aliasing of security relevant properties, substituting the original code with a wrapper that keeps the original reference and realizes the policy enforcement logic: this approach is known as *aspect-oriented programming*, in which a code snippet (*advice*) is executed at some point and under certain conditions (*pointcut*). For the developer this modification comes at the cost of the definition of custom policies and the inclusion of a JavaScript library, the already developed code remains unchanged. For an attacker is still possible to overwrite the reference to a built-in function to accomplish his malicious activity, however he cannot access the original function because the unique reference is held by the wrapper. The policy enforcement is based on an object containing the security state that represents interesting features of the action performed by the code so far. Using this construct is possible to define fine grained stateful security policies as done in [50]. The main issue of this method is that it quite easy for the developer to leave unprotected a security sensitive reference to a method due to the high number of access syntax provided natively by JavaScript: if a single reference is accessible to the attacker the entire security mechanism is bypassed; another important concern is the development of the policies that is entirely demanded to the developer without any support from the reference monitor.

Although employing the powerful paradigm of the aspect-oriented programming the reference monitor described in [53] lacks in providing a reliable security layer due to its implementation as a JavaScript library. To overcome this limitation is possible to realize the policy enforcement directly inside the browser which is in the best position to control the execution of JavaScript code (similarly to what described in section §5.5). This approach is implemented by ConScript [54] which modifies the JavaScript engine to add a native support to advice inclusion. ConScript modifies the heap structures that are used by the JavaScript engine to handle both native and user defined functions, in order to add a pointer to the advice responsible to enforce the policies associated to the specific function; with this modification every time a function is called the execution is diverted in executing the advice that then can decide to run the real function or not according to the policies. Modifying the browser JavaScript engine to intercept calls to a function at memory object level, the wrapping of all the access paths to the function made available by the JavaScript environment is not needed anymore, raising the bar for an attacker that tries to evade enforcement: the effectiveness of the security level provided by ConScript is completely entrusted to the correctness of the policies that must block every unwanted operation. Together to the deep advice implementation the integration with the browser JavaScript engine allows ConScript to run without adding considerable overhead to the execution: with respect to the approach in [53] that slows down operations by more than 200%, ConScript adds an overall overhead to the entire execution smaller than 10%.

The possibility to implement a reference monitor to secure a web application is the most desirable scenario for a web developer that can write code without the need of employing sandboxing framework constructs and then reuse legacy code. The policy development can be a non-trivial task for an unskilled developer and also an experienced one may leave unprotected some sensitive operations, to address this issue in [54] are described two techniques to automatically generate policies starting from the application code: the former based on static analysis and the latter which recurs to runtime learning to determine the expected behavior.

5.3 Language sandboxing

Most of the problems related to the secure inclusion of the JavaScript code in web pages derive from the use of native features of the JavaScript language that allows a developer to exert powerful operations that are uncommon for other programming languages (§ 2.1). These constructs allows for a deep level control of the execution flow making very difficult to statically analyze the code and in addition pose many obstacles to an efficient confinement as already highlighted in sections § 5.1 and § 5.2. A viable method to eradicate this problem is to forbid the use of dangerous language constructs, allowing for a more clear and predictable behavior of the scripts: this is the concept of language sandboxing. The approach of language sandboxing has already been implemented in popular websites like Facebook[4] and Yahoo![5]: in the former FBJS, a complete programming interface, is provided to developers to create applications while in the latter

[4]Facebook, `www.facebook.com`
[5]Yahoo!, `www.yahoo.com`

the ADsafe filter[6] defines a secure JavaScript subset for secure advertisement inclusion; anyway both of them present flaws that give chance to an attacker to compromise the hosting page [55, 56, 57, 58, 59, 60, 61]. The main issue in the development of a language sandboxing strategy is to find a trade-off between usability and security: the limitations introduced by the chosen subset of JavaScript should allow the legitimate applications to run correctly (and with the least possible modifications) while blocking security sensitive operations. The basis to build up such a protection mechanism is the analysis of the of the JavaScript language using a formal representation, this is usually done by constructing a small-step operational semantics [62] which exposes the properties of each construct and makes possible to prove the capabilities of the derived language subset.

Because existing benign scripts may doesn't comply with a subset of JavaScript the choice to restrict the language specification is unfeasible, anyway to overcome this issue is possible to enforce the fulfillment to the subset at runtime using a combination of filtering, rewriting and wrapping: filtering removes forbidden operations, rewriting translates allowed operations to comply with the subset of the language and inserts runtime checks to secure instructions that cannot be proved statically, finally wrapping protects environment properties from being corrupted. The approach described by Maffeis et al. in [56] starts from describing some interesting features of the JavaScript language, identifying those that can cause security problems, and then provide a set of code transformation that allows to obtain a safer version of the same code. The exposition lacks of a real implementation of the approach, giving only a comparison with FBJS, showing which are the improvements. The main restrictions imposed to the language concern the filtering of the keywords `eval`, `Function` and `constructor,` the rewriting of properties accesses using the bracket notation, the use of `this` and the wrapping of the `Object` and `Array` prototypes objects.

A similar approach that has an implementation in the real world is CAJA [63] a project developed by Google[7] to sanitize the included third-party code in web pages. The foundation of Google CAJA is the concept of object-capability language: in an object-capability language an object has the ability to interact with others only holding references, but in such a language objects doesn't have security sensitive references by default. Although JavaScript isn't an object-capability language Miller et al. discovered that a subset of the language has this property. The modifications imposed to the language concern (i) the restriction of the most constroversial constructs made available by the JavaScript specification (`eval` and `this`), (ii) limitations to access of certain

[6]ADsafe, `www.adsafe.org`
[7]Google, `www.google.com`

objects namespaces that are used to store environment information, (iii) creation of an ad-hoc environment for each imported module and (iv) the possibility to freeze an object making it immutable (all objects of the default global environment are frozen by default) ; given the common use of `eval` to de-serialize JSON objects this limited functionality is made available through an ad-hoc function. The interesting feature of CAJA is the possibility to both statically transform the code to comply with the object-capability subset or perform the translation at runtime in case of third-party scripts: both these operations are denoted as *cajoling*. To further harden defenses an additional subset of JavaScript has been developed: CAJITA; the only difference with CAJA is the absence of the keyword `this` that requires great effort to be supported and doesn't provide additional functionalities to the language. The main difference with respect to the approach in [56] is the possibility to include also policies similar to tool described in sections § 5.1 and § 5.2: this powerful feature doesn't come for free but causes a performance reduction and an increased code complexity. As a further refinement step of the effort for the development of Google CAJA Maffeis et al. demonstrated in [58] that the core language of CAJITA has the property called authority-safety that in turn implies complete isolation capabilities of objects with different authorities.

5.4 Information flow control

One of the possible target of attacks deriving from the unsafe inclusion of third-party code in web pages are credentials or sensitive user information. In this section a collection of methods is presented which have the aim to protect information from being stolen, most of the tools here described make use of taint analysis techniques to keep track of the spreading of information during the execution and block those flows that cause the exit of information from the browser boundaries or other unwanted scenarios. The other objective of the following tools is to grant not only the confidentiality but also the integrity of the information, disallowing third-party code to tamper with them.

Chugh et al. described in [10] a framework for the staged analysis of the information flow in a web page. Knowing that a complete dynamic analysis performed at client side would introduce an unacceptable runtime overhead, they used an approach that comprises a static analysis phase at server side which computes all the information flows available, adding to the code the logic responsible of enforcing flow policies at runtime when the embedded code is loaded; following this methodology also the dynamic generated code is subject to the analysis. The basis of the entire process is the definition of flow policies i.e. pairs of entities that represent source and destination of forbidden information flows: using such policies is possible to dictate where information can flow (confidentiality policy) or which operations can modify certain information (integrity policy); the enforcement of flow policies is based on the propagation of taints dynamically applied to the properties. Even if the approach is developed especially to keep the runtime overhead low, time measurement shows that as the code length increases the overhead can reach the magnitude of seconds.

A similar approach for the analysis of the information flow is described in [11]. Differently from the previous approach Jang et al. realize their policy enforcement dynamically rewriting the code to add a taint field to every object and the propagation and verification logic, in addition the policy language is richer because here is possible to specify which are the domains (i.e. sources of the third-party code) that have the permissions to read/write sensitive information. This information flow control system is implemented modifying the browser adding the code that automatically performs the rewriting before the script code gets sent to the JavaScript interpreter, in this way all the code is subject to the control (including the dynamic generated one). The benchmarks for this security mechanism give reason to the assumption of Chugh et al. that tried to reduce the overhead shifting the heavier part of the analysis to the server side, as a matter of fact the overhead imposed by the information flow control mechanism is in the order of second, reaching in certain case peaks of 6-7 seconds, unacceptable for the final user.

Beside the issue of browser slowdown the previous approaches don't cover an important part of the problem due to the presence of browser extensions written in JavaScript. These extensions amplifies all the problems that are described in chapter 4 and chapter 5 because of their position that allows them to run with the same level of privileges of the browser. Given this concern having a specialized tool that tries to solve the problem of controlling the information spreading inside browser extension would be useful.

Dhawan and Vinod in [23] describe SABRE a tool for the dynamic tracking of information inside JavaScript browser extensions. The concept that guides SABRE in detecting undesired information flows is the identification of sensitive sources and low-sensitive sinks, i.e. destinations where sensitive information should not flow: file system and network. The tool considers as sensitive sources every entity that has access to DOM elements or to the persistent storage and so every information coming from those sources is considered sensitive itself; another useful feature is the possibility to detect execution of untrusted code coming from a non-sensitive source (e.g. passing a string received from the network to `eval`). The tracking is performed by modifying the browser JavaScript engine, making possible to add to every object three fields that represent (i) a sensitivity level, (ii) a flag for the modification performed by a browser extension and (iii) the name of the extension that permormed the modification ; although this approach allows to track every type of information flow, the analysis is limited to those flow in which JavaScript browser extensions are involved. The main limitation in which SABRE incurs is the poorness of the taint analysis which depends from the use of a single taint field, but in contrast with previous approaches, that make use JavaScript instrumentation to enforce flow policies, the choice to modify the JavaScript engine allows to achieve better performance.

VEX [24] is a tool for the static offline verification of the information flow in browser extensions, its main usage is during the vetting of user submitted browser extension in online catalogs like Chrome Web Store[8]: the objective of the tool is to find dangerous information flows together with unsafe programming practices that can give place to vulnerabilities. The use of a tool like VEX should allow for a more precise and comprehensive analysis of JavaScript browser extension, dramatically reducing the runtime overhead shifting the analysis phase at server side. The analysis performed by VEX is based on the AST representation of the JavaScript code, that is navigated to extract all the interesting sources and sinks and to find instances of known bad programming practices; once sources and sinks have been identified the analysis proceed by using taint-analysis to find suspicious flows and in that case an alert is reported. Although the basic idea of shifting the flow analysis to the server before delivering the virtually vulnerable extension to the user is correct, the implementation has a big limitation: the analysis approach is not general as it searches only for a predefined set of information flow, possibly leaving undetected other dangerous flows.

Looking at the research efforts in the filed of controlling the information flow inside the browser is clear that none of the previous approaches solves entirely the problem, anyway future research should focus these guidelines:

- In the case of a tool realized to protect the user during the navigation it must be integrated with the browser because techniques that realize dynamic information flow control in JavaScript have demonstrated to introduce an unacceptable runtime overhead that affects browser performance.
- The analysis approach must be general, identifying all possible types of information flow by giving a general description of sources and sinks: in this way novel attack types can be countered effectively.

[8]Chrome Web Store, `https://chrome.google.com/webstore/`

- In the case of server-side analysis would be useful to implement a dynamic analysis strategy with different browser configurations because the hardware constraints are less limiting.

5.5 XSS prevention

The approaches described in section §4.5 have the aim to inhibit a JavaScript worm to propagate further, blocking operations that are used to post the worm in a web page exploiting a XSS vulnerability, unfortunately they do not provide protection against damages caused on the user machine by the execution of the injected script. Diversely the two following approaches have the objective to prevent the execution of injected scripts, giving a general solution to fight XSS attacks at client side.

BEEP [5] is a protection mechanism designed to prevent injected scripts execution via the enforcement of non-execution policies. The methodology is based on the observation that (i) the web browser in the best position to decide whether a script should be executed or not and (ii) the web page developer knows exactly which scripts are supposed to be present in the web page . These observations demonstrate that the collaboration between the browser and the web page developer can effectively discriminate about the execution of scripts embedded in a page. The approach taken by BEEP is to instrument the browser to consult a whitelist of benign scripts defined by the developer: whenever a script is about to be executed a hook function is called to 1) calculate the SHA-1 hash of the script, 2) look for a matching in the whitelist and 3) in the case of a positive match execute the script . This protection mechanism is combined with another security measure (based on blacklisting) that allows a page developer to specify that an entire branch of the DOM tree is not executable: doing this each script contained in the branch won't be run. The main issue that must be addressed to make this approach tamper proof is to grant BEEP code to be the first script that gets executed, in this way every subsequent script will be subject to the whitelisting/blacklisting filtering, making impossible for an attacker to evade the policy enforcement: to satisfy this condition the library is inserted as the first script in the `head` tag of the page[9]. Beside the effectiveness of the method it's noteworthy that its application requires deep modifications in the rendering routine and in the JavaScript engine, leaving unprotected older versions of the browsers.

An analogous approach based on collaboration between server and browser is imple-

[9]This is the result of an empirical observation, because the HTML standard doesn't specify an ordering among the scripts

mented in BLUEPRINT [6] that instead of enforcing policies in the browser allows the web application to take control of the parsing of the untrusted content of the page. The purpose of BLUEPRINT is to allow a web application to support complex HTML-based, user-generated (and so untrusted) content while providing protection against residual JavaScript code that didn't get filtered. The methodology employed by BLUEPRINT requires a processing stage at server side that creates a parsing tree starting from each chunk of untrusted content, pruning nodes that contain JavaScript code and other active components (e.g. Flash object); the tree is then embedded in the page together with a parsing routine that will reconstruct the filtered content. The parsing routine is the core of the protection mechanism at client side, to not hamper the entire process it has to grant that during the reconstruction of the untrusted content no functions that can cause code evaluation are ever called: this is done in practice with common DOM APIs for which this property has been demonstrated. To add a further layer of protection the parse tree is also encoded using the Base64 scheme that makes it *syntactically inert*, reducing the risk of misinterpretation that can lead to code execution. The implementation of BLUEPRINT doesn't require any browser modification and also server-side deployment is quite straightforward (it consists in a PHP library inclusion); in addition, being employed only native DOM APIs for the client-side processing, the protection mechanism can be added also to older version of the browsers.

Comparing the approaches described here with those reported in section §4.5 is clear that using a defense mechanism based on collaboration between server and browser is possible to achieve a more effective protection to both standard XSS attacks and JavaScript worms. The former provide a more general solution that in addition to block the worm propagation denies their execution; implementing such approaches is straightforward, doesn't require additional infrastructure at server side and requires only minor changes in the web browser (made easy realizing a proper browser extension). A desirable scenario would be to define a widely accepted standard for browser collaboration implemented by the major web sites, driving web browsers developers to the integration of the approach in their products, maybe extending the expressive power of the CSP (§ 2.2.2).

Chapter 6

Conclusions

In this thesis has been reported a systematization of knowledge concerning the presence of threats deriving from a malicious use of JavaScript code, that makes the web browsing a dangerous operation for the user, if performed without proper protection mechanisms. The analysis showed that the problems derive from these intrinsic characteristics of the JavaScript language:

1. An unsafe script inclusion paradigm that allows malicious code to run with the same privileges of the native code

2. Wide support for code obfuscation strategies, that eases the attacker in hiding its malicious code.

The systematization then provides a characterization of attacks that identifies the different vectors and the most employed strategies. This analysis makes possible to identify which are the "parts" of the problem to be addressed. Comparing different security measures described in literature it was possible to realize a global taxonomy of approaches in addition to the identification of strengths and weaknesses of each of them. Using the taxonomy of the defense measures and the aforementioned model of the attacks it was then realized a matching between attacks and corresponding countermeasures that highlights which are the areas that are still open for future works as described in section §6.1.

6.1 Future works

The description of the current defenses that are implemented in common user's machines shows clearly that they are not able to provide a sufficient level of protection, making necessary the implementation of other approaches. Looking at the characterization of the attacks strategies described in chapter 3 is possible to notice that tools described in literature are able to provide an higher level of protection by countering the attacks at different stages. Unfortunately there are some issues that are not completely solved and concern the possibility for the malicious code to detect the presence of a defense tool and the act accordingly stopping the attack or trying to evade the security measure.

Having analyzed different classes of security measures the future research efforts should be directed in realizing:

- A detection mechanism for malicious JavaScript code directly integrated in the browser. The basic approach that should be chosen as motivating example is IceShield[8] that provides a reliable detection, being able of analyze all the code belonging to a web page. The integration in the browser is an indispensable feature that allows to reach higher performance (thus countering certain fingerprinting techniques) and to have a full coverage over all the pages opened in the browser.

- A lightweight information flow control system that protects users' sensitive information from being stolen. Also in this case the low-level integration in the web browser is needed, because the main limitation of these type of tool is the runtime overhead imposed to the execution of the scripts.

Bibliography

[1] M. Egele, E. Kirda, and C. Kruegel, “Mitigating drive-by download attacks: Challenges and open problems,” *Proceedings of Open Research Problems in Network Security Workshop (iNetSec)*, pp. 1–11, Jan. 2009.

[2] F. Maggi, A. Frossi, S. Zanero, G. Stringhini, B. Stone-Gross, C. Kruegel, and G. Vigna, “Two Years of Short URLs Internet Measurement: Security Threats and Countermeasures,” in *Proceedings of the 22Nd International Conference on World Wide Web.* Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2013, pp. 861–872.

[3] M. Cova, C. Kruegel, and V. Giovanni, “Detection and analysis of drive-by-download attacks and malicious JavaScript code,” in *Proceedings of the 19th international conference on World wide web.* New York, NY, USA: ACM, 2010, pp. 281–290.

[4] L. Invernizzi and P. M. Comparetti, “EvilSeed: A Guided Approach to Finding Malicious Web Pages,” *Proceedings of the 33th IEEE Symposium on Security and Privacy*, pp. 428–442, May 2012.

[5] T. Jim, N. Swamy, and M. Hicks, “Defeating script injection attacks with browser-enforced embedded policies,” in *Proceedings of the 16th international conference on World Wide Web.* ACM, May 2007.

[6] M. ter Louw and N. V. Venkatakrishnan, “ Blueprint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers ,” *Proceedings of the 30th IEEE Symposium on Security and Privacy*, 2009.

[7] M. S. Miller, J. Nagra, and J. Terrace, “Cujo: efficient detection and prevention of drive-by-download attacks,” in *Proceedings of the 26th Annual Computer Security Applications Conference.* New York, NY, USA: ACM, 2010, pp. 31–39.

[8] M. Heiderich, T. Frosch, and H. Thorsten, “IceShield: detection and mitigation of malicious websites with a frozen DOM,” in *Proceedings of the 14th international conference on Recent Advances in Intrusion Detection.* Berlin, Heidelberg: Springer-Verlag, 2011, pp. 281–300.

[9] C. Curtsinger, B. Livshits, Z. Benjamin, and C. Seifert, “ZOZZLE: fast and precise in-browser JavaScript malware detection,” in *Proceedings of the 20th USENIX conference on Security.* Berkeley, CA, USA: USENIX Association, 2011, pp. 3–3.

[10] R. Chugh, J. A. Meister, J. Ranjit, and S. Lerner, “Staged information flow for javascript,” in *Proceedings of the 14th ACM conference on Programming language design and implementation.* New York, NY, USA: ACM, 2009, pp. 50–62.

[11] D. Jang, R. Jhala, S. Lerner, and H. Shacham, “Rewriting-based dynamic information flow for JavaScript,” *2010*, Jan. 2010.

[12] C. Kolbitsch, B. Livshits, Z. Benjamin, and C. Seifert, “Rozzle: De-cloaking Internet Malware,” in *Proceedings of the 33th IEEE Symposium on Security and Privacy.* Washington, DC, USA: IEEE Computer Society, 2012, pp. 443–457.

[13] A. Kapravelos and U. Barbara, “Revolver: An Automated Approach to the Detection of Evasive Web-based Malware,” *Proceedings of the 22nd USENIX Security Symposium*, Jan. 2013.

[14] S. R. Beard, N. P. Katta, Z. Tzermias, G. Sykiotakis, M. Polychronakis, and E. P. Markatos, "A Symbolic Execution Framework for JavaScript," in *Proceedings of the 31th IEEE Symposium on Security and Privacy.* Washington, DC, USA: IEEE Computer Society, 2010, pp. 513–528.

[15] F. Howard, "Malware with your Mocha," 2010, pp. 1–18.

[16] B. Feinstein, D. Peck, and I. SecureWorks, "Caffeine monkey: Automated collection, detection and analysis of malicious javascript," *Black Hat USA*, Jan. 2007.

[17] M. Rajab, L. Ballard, and N. Jagpal, "Trends in circumventing web-malware detection," *Google*, pp. 1–12, Jan. 2011.

[18] G. Lu and S. Debray, "Weaknesses in defenses against web-borne malware," *Proceedings of the 10th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Jan. 2013.

[19] D. Canali, M. Cova, G. Vigna, and C. 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.* New York, NY, USA: ACM, 2011, pp. 197–206.

[20] G. Lu and S. Debray, "Automatic Simplification of Obfuscated JavaScript Code: A Semantics-Based Approach," in *Proceedings of the 6th IEEE Sixth International Conference on Software Security and Reliability.* Washington, DC, USA: IEEE Computer Society, 2012, pp. 31–40.

[21] A. Kapravelos, M. Cova, K. Christopher, and G. Vigna, "Escape from monkey island: evading high-interaction honeyclients," in *Proceedings of the 8th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment.* Berlin, Heidelberg: Springer-Verlag, 2011, pp. 124–143.

[22] A. Ikinci, T. Holz, and F. Freiling, "Monkey-spider: Detecting malicious websites with low-interaction honeyclients," in *Proceedings of Sicherheit*, Jan. 2008.

[23] M. Dhawan and V. Ganapathy, "Analyzing Information Flow in JavaScript-Based Browser Extensions," in *Proceedings of the 25th Annual Computer Security Applications Conference.* IEEE Computer Society, Dec. 2009.

[24] S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett, "VEX: vetting browser extensions for security vulnerabilities," in *Proceedings of the 19th USENIX conference on Security.* Berkeley, CA, USA: USENIX Association, 2010, pp. 22–22.

[25] Y. Cao, V. Yegneswaran, P. Possas, and Chen, "Pathcutter: Severing the self-propagation path of xss javascript worms in social web networks," in *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)*, San Diego, California, USA, 2012.

[26] B. Livshits and W. Cui, "Spectator: detection and containment of JavaScript worms," in *Proceedings of the USENIX Annual Technical Conference.* Berkeley, CA, USA: USENIX Association, 2008, pp. 335–348.

[27] A. Dewald, T. Holz, and F. Felix, "ADSandbox: sandboxing JavaScript to fight malicious websites," in *Proceedings of the 25th ACM Symposium on Applied Computing.* New York, NY, USA: ACM, 2010, pp. 1859–1864.

[28] M. Egele, P. Wurzinger, K. Christopher, and E. 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.* Berlin, Heidelberg: Springer-Verlag, 2009, pp. 88–106.

[29] P. Ratanaworabhan, B. Livshits, and B. Zorn, "NOZZLE: a defense against heap-spraying code injection attacks," in *Proceedings of the 18th conference on USENIX security symposium.* Berkeley, CA, USA: USENIX Association, 2009, pp. 169–186.

[30] K. Selvaraj and N. F. Gutierrez, "The rise of PDF malware," *Symantec Security Response*, Jan. 2010.

[31] D. Stevens, “ Malicious PDF Documents Explained ,” *IEEE Security & Privacy*, vol. 9, no. 1, pp. 80–82, 2011.

[32] Z. Tzermias, G. Sykiotakis, M. Polychronakis, and E. P. Markatos, “Combining static and dynamic analysis for the detection of malicious documents,” in *Proceedings of the 4th European Workshop on System Security.* ACM, Apr. 2011.

[33] P. Laskov and N. S, “Static detection of malicious JavaScript-bearing PDF documents,” in *Proceedings of the 27th Annual Computer Security Applications Conference.* New York, NY, USA: ACM, 2011, pp. 373–382.

[34] C. Smutz and A. Stavrou, “Malicious PDF detection using metadata and structural features,” in *ACSAC ’12: Proceedings of the 28th Annual Computer Security Applications Conference.* ACM, Dec. 2012.

[35] D. Maiorca, I. Corona, and G. Giacinto, “Looking at the bag is not enough to find the bomb: an evasion of structural methods for malicious PDF files detection,” in *ASIA CCS ’13: Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security.* ACM, May 2013.

[36] K. Schütt, M. Kloft, A. Bikadorov, and K. Rieck, “Early detection of malicious behavior in JavaScript code,” in *Proceedings of the 5th ACM workshop on Security and artificial intelligence.* ACM, Oct. 2012.

[37] L. Lu, V. Yegneswaran, P. Porras, and W. Lee, “BLADE: an attack-agnostic approach for preventing drive-by malware infections,” in *Proceedings of the 17th ACM conference on Computer and communications security.* ACM, Oct. 2010.

[38] J. Zhang, C. Seifert, J. W. Stokes, and W. Lee, “ARROW: Generating Signatures to Detect Drive-By Downloads,” in *WWW ’11: Proceedings of the 20th international conference on World wide web.* ACM, Mar. 2011.

[39] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, and S. King, “Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites That Exploit Browser Vulnerabilities,” in *Proceedings of the 13th Network and Distributed System Security Symposium (NDSS).* San Diego, California, USA: NDSS, 2006, pp. 1–15.

[40] J. Nazario, “PhoneyC: a virtual client honeypot,” in *Proceedings of the 2nd USENIX conference on Large-scale exploits and emergent threats: botnets spyware, worms, and more.* Berkeley, CA, USA: USENIX Association, 2009, pp. 6–6.

[41] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, “You are what you include: large-scale evaluation of remote javascript inclusions,” in *Proceedings of the 19th ACM conference on Computer and communications security.* ACM, Oct. 2012.

[42] M. ter Louw, K. T. Ganesh, and N. V. Venkatakrishnan, “AdJail: practical enforcement of confidentiality and integrity policies on web advertisements,” in *Proceedings of the 19th USENIX conference on Security.* Berkeley, CA, USA: USENIX Association, 2010, pp. 24–24.

[43] Z. Luo and T. Rezk, “ Mashic Compiler: Mashup Sandboxing Based on Inter-frame Communication ,” *Proceedings of the 25th IEEE Computer Security Foundations Symposium*, 2012.

[44] A. Barth, C. Jackson, and W. Li, “Attacks on javascript mashup communication,” in *Proceedings of Web 2.0 Security and Privacy*, Jan. 2009, pp. 1–8.

[45] S. Crites, F. Hsu, and H. Chen, “OMash: enabling secure web mashups via object abstractions,” in *Proceedings of the 15th ACM conference on Computer and communications security.* ACM, Oct. 2008.

[46] J. Mickens and M. Finifter, “Jigsaw: efficient, low-effort mashup isolation,” in *Proceedings of the 3rd USENIX conference on Web Application Development*, Jan. 2012.

[47] X. Dong, M. Tran, Z. Liang, and X. Jiang, “AdSentry: comprehensive and flexible confinement of JavaScript-based advertisements,” in *Proceedings of the 27th Annual Computer Security Applications Conference.* ACM, Dec. 2011.

[48] S. Bandhakavi, S. T. King, and P. Madhusudan, “JavaScript in JavaScript (js.js): sandboxing third-party scripts,” in *Proceedings of the 3rd USENIX conference on Web Application Development.* Berkeley, CA, USA: USENIX Association, 2012, pp. 9–9.

[49] L. Ingram and M. Walfish, “TreeHouse: JavaScript sandboxes to helpWeb developers help themselves,” in *Proceedings of the 2012 USENIX conference on Annual Technical Conference.* Berkeley, CA, USA: USENIX Association, 2012, pp. 13–13.

[50] P. Agten, S. van Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens, “JSand: complete client-side sandboxing of third-party JavaScript without browser modifications,” in *Proceedings of the 28th Annual Computer Security Applications Conference.* New York, NY, USA: ACM, 2012, pp. 1–10.

[51] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir, “BrowserShield: vulnerability-driven filtering of dynamic HTML,” in *Proceedings of the 7th symposium on Operating systems design and implementation.* Berkeley, CA, USA: USENIX Association, 2006, pp. 61–74.

[52] J. A. Meister, J. Ranjit, S. Lerner, and M. Cova, “JavaScript instrumentation for browser security,” in *Proceedings of the 34th ACM symposium on Principles of programming languages.* New York, NY, USA: ACM, 2007, pp. 237–249.

[53] P. H. Phung, D. Sands, and A. Chudnov, “Lightweight self-protecting JavaScript,” in *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security.* New York, NY, USA: ACM, 2009, pp. 47–60.

[54] L. A. Meyerovich and B. Livshits, “ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser,” in *Proceedings of the 31th IEEE Symposium on Security and Privacy.* Washington, DC, USA: IEEE Computer Society, 2010, pp. 481–496.

[55] S. Maffeis and A. Taly, “Language-based isolation of untrusted Javascript,” in *22nd IEEE Computer Security Foundations Symposium.* IEEE: IEEE, 2009, pp. 77–91.

[56] S. Maffeis, J. C. Mitchell, and T. Ankur, “Isolating JavaScript with filters, rewriting, and wrappers,” in *Proceedings of the 14th European conference on Research in computer security.* Berlin, Heidelberg: Springer-Verlag, 2009, pp. 505–522.

[57] S. Maffeis, J. Mitchell, and A. Taly, “Run-time enforcement of secure javascript subsets,” *Proceedings of Web 2.0 Security and Privacy*, pp. 1–19, Jan. 2009.

[58] S. Maffeis, J. C. Mitchell, and A. Taly, “ Object Capabilities and Isolation of Untrusted Web Applications ,” *Proceedings of the 31st IEEE Symposium on Security and Privacy*, 2010.

[59] M. Finifter, J. Weinberger, and A. Barth, “Preventing capability leaks in secure javascript subsets,” in *Proceedings of the 17th Network and Distributed System Security Symposium (NDSS)*, 2010, pp. 1–14.

[60] J. G. Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi, “ADsafety: type-based verification of JavaScript Sandboxing,” in *Proceedings of the 20th USENIX conference on Security.* Berkeley, CA, USA: USENIX Association, 2011, pp. 12–12.

[61] C. Kruegel, G. Vigna, C. Anderson, P. Giannini, and S. Drossopoulou, “Automated Analysis of Security-Critical JavaScript APIs,” in *Proceedings of the 32th IEEE Symposium on Security and Privacy.* Washington, DC, USA: IEEE Computer Society, 2011, pp. 363–378.

[62] S. Maffeis, J. C. Mitchell, and T. Ankur, “An Operational Semantics for JavaScript,” *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, pp. 307–325, 2008.

[63] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay, “Caja Safe active content in sanitized JavaScript,” Unknown, Ed., Jan. 2008.