Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Malicious JavaScript Analyser

Author:
Hongtao Li

Supervisor:
Dr. Sergio Maffeis

Second Marker:
Dr. Mahdi Cheraghchi

May 11, 2018

Abstract

Your abstract goes here

Acknowledgements

Thanks mum!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

JavaScript is highly dynamic, expressive language that supports real time evaluation and dynamic code generation at runtime. Today, JavaScript are not only been used in the development of large-scale web site such as Facebook but also used in different browser extensions. However the expressiveness and dynamic feature of JavaScript is often misused by attackers. The most widely used technique is the use of **eval** and **document.write** to convert runtime strings into invocable code.

In recently years, the number of JavaScript based malware attacks have increased. By exploiting numerous vulnerabilities in various web applications, attackers can launch a wide range of attacks such as cross-site scripting(XSS)[**?**], cross-site request forgery(CSRF)[**?**], drive-by downloads [**?**], etc. And most Internet users rely on the anti-virus software. However, most existing anti-virus software use static signature to prevent malicious JavaScript being executed. But attacker usually applies different JavaScript obfuscation techniques in order to hide the malicious content to evade detection.

Apart from the language features, *exploit kits*[**?**] allow attackers to create new exploits by using multiple existing exploits. Depending on the change, this can be easily accomplished within minutes. However, this also means there exists certain patterns behind the exploits.

1.1 Objectives

In order to provide a reliable detection for malicious JavaScript, this project is to build a malicious JavaScript code Analyser that can not only classify a piece of JavaScript code into an known malicious class but also analysis the report malicious JavaScript patterns used.

Malware cycle is an asymmetry loop between attackers and defenders. Once attackers develop the new malicious codes, they can always test it with the existing anti-virus engines until it can evade the detection. This project should be able to provide information about the relations between different patterns. In order to let the defenders to have a better understanding on how the patterns evolve with time.

1.2 Challenges

JavaScript is embedded in web pages, except the script blocks, JavaScript codes can also exists in event handlers (e.g. button onclick event). Some malicious content may only be generated based on those events which means the malicious content can be hidden on the static perspective. So correctly detect a piece of JavaScript is malicious or not is the biggest challenge.

1.3 Contributions

...

Chapter 2

Background

2.0.1 JavaScript Obfuscation Techniques

In order to find the malicious JavaScript patterns, we need to study the usage of different JavaScript obfuscation techniques in a systematic way. The following are the six categories based on different operations, more details can be found in[**?**]:

1. **Randomisation Obfuscation**:

   (a) Insert some elements of JavaScript code such as white space [**?**] and comments. (White space includes space character, tab, line feed, form feed and carriage return.) Because of JavaScript interpreters ignore white space characters and comments these changes won't affect the semantics of the code.

   (b) Randomly replace variable and function name.

   Attackers usually combine the above two randomise techniques together to increase the chance of evading the detection. Following code snippets have exactly same semantics but different static names.

   [language=JavaScript, title=(original code)] function myFunction(name) alert("Hello " + name); var myName = "world"; myFunction(myName);

   [language=JavaScript, title=(using randomisation)] function $_0xa88xj1(_0x94e9x2)alert('Hello' +_0 x94e9x2)///rando$ $world';_0 xa88xj1(_763kkdi)$

2. **Number Obfuscation**:
   Same number can be expressed in different ways by using basic arithmetic operations. (e.g. var x = 10, var x = 5+5, var x = 1000/100, etc.)

3. **String Obfuscation**:
   String obfuscation techniques are widely used when people want to obfuscate JavaScript codes. Apart from some special string obfuscation based on JavaScript language behaviours. (e.g. "(![]+[])[+[]]+(![]+[])[+!+[]]+([![]]+[][[]])[+!+[]+[+[]]]+(![]+[])[!+[]+!+[]]" will be evaluated to string "fail"). The following are the two main categories of string obfuscation.

   (a) **Encoding**:
   Encoding change the presentation of a string makes the code hard to interpret by humans but doesn't change the actual meaning of the string. This technique can be used to protect code privacy or intellectual property as well as to evade detection.

      i. URL Encoding(% Encoding)
      ii. Unicode Encoding
      iii. Customised Encoding Function (decode during execution)

   [language=JavaScript, title=(encoding examples)] var $url_str = unescape("\%68\%65\%6C\%6C\%6F\%20\%77\%$ $unescape("\%u0068\%u0065\%u006C\%u006C\%u006F\%u0020\%u0077\%u006F\%u0072\%u006C\%u0064"); var$ $"helloworld";$

   (b) **String Manipulation**:
   Attackers could change the order of sub-strings and assign them with randomly generated variable names to make the code less human readable.

i. Concatenation: split a string into the concatenation of several sub-strings. Because of the dynamic feature of JavaScript, this usually used along eval() or document.write() to executed the concatenated string.

ii. Character Substitution: usually used with the replace() function and regular expression to substitute some of the characters in the given string before executing

iii. Keyword Substitution: use a variable to substitute JavaScript keywords.

[language=JavaScript, title=(concatenation)] var t2="ri"+"te"+"("+"¨"; var t3="hello"+"world"+"¨"+");"; var t1="doc"+"um"+"ent"+"."+"w"; eval(t1+t2+t3);

[language=JavaScript, title=(character substitution)] var str="!@h@e *l)l++ow?o/rld"; doc- u- ment.write(str.replace(/[a− zA− Z0− 9]/g,''));

[language=JavaScript, title=(keyword substitution)] var test=document; test.write("helloworld");

[language=JavaScript, title=(original code)] document.write("helloworld");

Obfuscated Field Reference:
JavaScript allows an object being accessed in two different ways:

1. object.field
2. object["field"]

So that the index expression may be computed and using string obfuscation stated above to obfuscate the code.

[language=JavaScript, title=(obfuscated field reference example)] document["write"]("helloworld");

document.write("helloworld");

var b = ['obj1', 'obj2']; b.obj1 = "test1"; b.obj2 = "test2"; b['obj1']['replace'](/1/,'2')]; //"test2"

Logic Structure Obfuscation[?]:
Attackers may change the logic structure to manipulate the execution paths of JavaScript codes. This won't affect the original semantics.

1. insert some instrucions which are independent to the functionality
2. add/change some conditional branches

[language=JavaScript, title=(logic structure obfuscation example)] var i = 100; if (i < 0) alert("never!"); // dead code for (i = 0; i < 100; i++) if (i == 50) document.write("helloworld"); The above example first inserts the independent instructions "if (i<10) {...}" which are dead codes, used to complicate the execution path check. Then add an extra conditional branch inside the for loop "if (i==50) {...}" which will be executed.

Environment Interactions[?]:
JavaScript is embedded in web pages to be run for clients within a web browser. The DOM allows several types of obfuscation can be applied. Attackers can scatter the JavaScript codes across the HTML page using multiple <script>blocks to make it harder to reverse engineer (one script can be splitted into several script blocks or even loaded remotely from a file).

[language=JavaScript,alsolanguage=HTML5, title=(self-contained block)] <script> var a="helloworld"; alert(a); </script>

[language=JavaScript,alsolanguage=HTML5, title=(multiple script blocks)] <script> var a="helloworld"; </script> ... <script> alert(a); </script>

[language=JavaScript,alsolanguage=HTML5, title=(remote source file)] <script src="helloworld.js"> </script> ... <script> alert(a); </script> [helloworld.js] var a="helloworld";

JavaScript can also be hidden in environment events. (e.g. button onclick event, etc.) This prevents the malicious code being detected by simply extracting the <script>blocks from the page. [language=JavaScript,alsolanguage=HTML5, title=(script in events)] <!DOCTYPE html> <html> <body> <button onclick="(function f() alert('helloworld');)()">myButton</button> </body> </html>

2.0.2 Benign and Malicious JavaScript

JavaScript obfuscation techniques can be used in benign code as well. In order to protect code privacy or intellectual property, company will using some obfuscation techniques to reduce the readability of their JavaScript codes. Tool-kits such as JavaScript Obfuscator [**?**] can be used to make the codes less human unreadable. So obfuscation doesn't imply malicious. We can't say a piece of code is malicious just because it is obfuscated.

Malicious JavaScript code exploits obfuscation to hide its malicious intent to evade the detection and doesn't care the performance of the code after obfuscation while for benign codes they won't downgrading the execution performance by applying obfuscation.

2.0.3 Existing Approaches

Static Analysis

The static analysis method checks the static characteristics and the structure of the JavaScript. But, exploits are frequently buried under multiple levels of JavaScript **eval**. Static analysis need to be performed on unfolded JavaScript codes to achieve better result. However, in order for a piece of JavaScript code being interpreted or compiled for execution, it needs to be decomposed into lexical tokens. The static analysis takes the advantages of this process via using the JavaScript lexical tokens directly without executing the script itself. This approach has negligible runtime overhead so it is widely used in browser extensions to block the malicious web pages. 2 [language=json,title=(tokens from Esprima)] "type": "Keyword", "value": "var", "range": [0, 3] , "type": "Identifier", "value": "x", "range": [4, 5] , "type": "Punctuator", "value": "=", "range": [6, 7] , "type": "Numeric", "value": "1", "range": [8, 9] , "type": "Punctuator", "value": ";", "range": [9, 10] The code will be decomposed into Keywords, Punctuators, Identifiers and Literals follows the language specification of JavaScript [**?**] sequentially.

For example, Esprima[**?**] is a high performance, standard-compliant JavaScript parser, if we use Esprima to parse "var x = 1" will produce the tokens on the left. We can see the type, value and the range of each token.

Based on those tokens, then we can provide further analysis based on rules and heuristics. For example, Support Vector Machines(SVM) based static analysis. Given benign and malicious codes as two sets of training data, an SVM can determines a hyperplane that separates both classes with maximum margin. After training, when feeding an unknown data to the SVM, it will map it to the vector space and classify it to either begin or malicious side of the hyperplane.

Since the actual name of variable or function doesn't change the semantic of the code itself, in Cujo[**?**]'s implementation those identifier tokens were replaced to the generic token **ID**. Similarly generic tokens **NUM** and **STR** are used instead of numerical literals and string literals respectively. To further strengthen the static analysis, they also record the length of each string literals (**STR.01** refers to a string with up to 10^1 characters) and added **EVAL** in their tokens for feature extraction.

[language=JavaScript, title=(original code)] var x = 1; var y = "helloworld"; var z = x + 15;

[language=JavaScript, title=(Cujo's implementation)] ID = NUM; ID = STR.01; ID = ID + NUM;

And then based on the concept of q-grams to perform feature extraction where q is the length of each pattern (i.e. number of consecutive tokens). This allows us to find the top feature of a certain attack with the corresponding tokens.

Another approach for feature extraction is based on the hierachical structure of JavaScript abstract syntax tree (AST) which is used in ZOZZLE[**?**]. In ZOZZLE's implementation, a feature contains two things: the context which it appears (e.g. loops, conditional branches, try catch blocks, etc.) and the text of the AST node. 2 [language=json,title=(AST from Esprima)] Program body[1] |–VariableDeclaration |–declarations[1] |–VariableDeclarator | |–id | |–Identifier | |–name:x | |–init | |–Literal | |–value:1 | |–raw:1 kind:var Their implementation limits the possible number of features for a better performance. Only add to the feature set if the AST node is expression or variable declaration. Then using **Bayesian classifier** to run the classifier training.

Left figure is a sample AST of a single variable declaration with initial value 1 (extracted from Esprima for "var x = 1"). They also add some pre-defined string patterns to speed up the matching process. All purely statical based detector will fail to detect some attacks if the malicious pattern doesn't match any of the known features. And theses kind of statical analysis tools need to be kept trained with new evading techniques to continue to be effective.

Environment Analysis

Web-based malware tends to be environment-specific which will attempt to fingerprint the version of the victim's software, for example, the browser and version of installed plug-ins. Following are the three main techniques attackers commonly used:

- **Environment matching**: the malicious JavaScript determines the capabilities of the browser and selectively alerts the content of the page.
- **Fingerprinting**: use a set of environment variables so that it is more comprehensive and detailed in its assessment.
- **Cloaking**[**?**]: is a technique that allow the malicious JavaScript code to have different behaviours (show different content) depends on who is visiting the page.

Malwares are triggered infrequently, which is the fundamental limitation for detecting a piece of code is malicious. It only reveal itself when running in the specific environment. [language=JavaScript,title=(Example JavaScript that checks for specific environment)] var obj = null; try obj = new ActiveXObject("AcroPDF.PDF"); catch (e) if (!obj) try obj = new ActiveXObject("PDF.PdfCtrl"); catch (e) if (obj) document.write('<embed src="exploits/x18.php..." type="application/pdf" width=100 height=100></embed>'); Rozzle[**?**] focus on the environment analysis that explores multiple environment related paths within a single execution. Their goal is to increase the effectiveness of dynamic crawler searching for malware.

Static analysis techniques that using AST can be performed to determining what conditions (e.g. if, try catch, etc.) in JavaScript code are environment-dependent.(focusing on **ActiveXObject** calls and **navigator** object)

Function Invocation Based Analysis

Many of the obfuscation techniques stated above works because of the dynamic generation and runtime evaluation feature of JavaScript. However, those only works with the help of theses functions:

1. JavaScript built in functions (e.g. **eval**, **usescape**)

2. DOM methods (e.g. **document.write**)

Apart from those built in functions. Due to the dynamic feature of JavaScript, user defined functions can also be invoked in multiple ways which also increase the difficulty of static checks.

```
[language=JavaScript, title=(function plus1)]
// global defined function function plus1(a)
return a+1;
[language=JavaScript, title=(passed as object
field)] var myArray = new Array(plus1, 1);
myArray[0](myArray[1]); //2
```

```
[language=JavaScript, title=(passed as array
element)] function funObj() this.f = plus1;
(new funObj).f(2); //3
[language=JavaScript, title=(passed as vari-
able)] var myVar = plus1; myVar(3); //4
```

The above example shows 3 different ways of invoking the same function. (function plus1 is predefined and in global scope).

JStill[**?**] focus on those malicious function arguments. Those malicious functions are called in a way that can hide their arguments from the static perspective. However, since both benign and malicious JavaScript codes using these techniques, the challenge is how to distinguish them.

Dynamic Analysis

Because not all code can be statically observed, for example the code hidden within **eval** string. Another direction of malicious code detection is based on the dynamic analysis which will actually run the code and try to cover as many code paths as possible to trigger the malicious part.

However, as mentioned in the previous section, attackers usually hide the malicious code using *cloaking*[**?**] techniques (only revealing the malicious content when the victim is using a specific version of the browser with a vulnerable plug-in). Therefore code coverage becomes the biggest challenge for dynamic analysis tools.

A successful dynamic analysis tool must have a large code coverage (same code must be run within all combination of the browsers and plugins) in order to detect malicious content efficiently. Call back feature of JavaScript is also difficult to capture, attackers can load the attack code only when a specific mouse lick event is triggered. (More details about will be explained in GUI exploration section)

Symbolic Execution: This technique is used to analysis a program to determine what inputs cause each part of the program execute (branches of code).

In dynamic symbolic execution, user inputs are treated as symbolic variables. Dynamic symbolic execution differs from normal execution in that while many variable have their concrete values like 1 for an integer variable, the values of other variables which depend on symbolic inputs are represented by *symbolic* formulas over the symbolic inputs, like $userinput+1$. Whenever any of the operands of a JavaScript operation is symbolic, the operation is simulated by creating a formula for the result of the operation in terms of the formulas for the operands.

For example:

Assume x has symbolic value $input_1 + 1$.
For an assignment operation $y = x$:
the symbolic execution of the operation copies this value to y. ($y = input_1 + 1$)

For an arithmetic operation $y = x + 5$:
the concrete values are calculated and symbolic part keep the same ($y = input_1 + 6$)

(String and boolean are treated in the similar way)

However, symbolically executing all feasible code paths does not scale to large application. The number of paths grows exponentially with an increase in program size. Therefore most tools that have symbolic executions generally use heuristics for path finding to reduce the execution cost and some use depth-limit to restrict the number of depths of execution performs.

Dynamic Symbolic Interpreter Before dynamic symbolic execution, the first step is to record the execution of the program with concrete inputs. JASIL[**?**] is an existing instrumentation component implemented in the web browser's JavaScript interpreter that can be used to record the semantics of the operations. It will capture all operations on integers, booleans, strings, arrays, as well as control-flows, object types, and calls to browser-native methods.
Once we have the recorded instructions we can run a symbolic interpreter to perform dynamic symbolic execution.

Path Constraint Extractor A concrete boolean value (true or false) will be recorded along each control-flow branch (e.g. if and else) during the execution for indicating if the branch was taken. In symbolic execution, the corresponding branch condition is recorded by the path constraint extractor if it is symbolic. [language=JavaScript, title=(example path constrains)] function checkNum(num) if (num $>$ 0) // (num $>$ 0) if (num $<$ 3) return "small"; // (num $>$ 0) AND (num $<$ 3) else return "big"; // (num $>$ 0) AND (num $>$ 3) return "error"; // (num $<$ 0) Path constraint is the formula formed by conjoining the symbolic branch conditions (negating the conditions if branches that were not taken) as execution continues. If an input value satisfies the path constraint, then the program execution on that input will follow the same execution path.

In the above example, if we take 2 as input which is greater than - and smaller than 3. Follow the path constraint for this input, we know the return value will be "small". [language=JavaScript] var myNum = 2; // (num $>$ 0) AND (num $<$ 3) checkNum(myNum); // "small" Based on the path constrains, we can use constraint solver to perform symbolic executions on the application.

Multi-execution: The idea of multi-execution is to execute the program multiple times with different input values in order to discover how inputs affect the behaviour of the program. While Rozzle[**?**] introduces single-pass multi-execution approach which execute both possibilities whenever it encounters control flow branching that is dependent on the environment. For example, in the case of if statement, both if and else branches will be executed. A key insight is to perform *weak updates*. Assignments in different branches while execution will only update the original value which means multi-execution won't cause dependency issue.

[language=JavaScript, title=(original code)] var a="hello"; var env=navigator.plugins[0].name; if (env=="Chrome PDF Plugin") a+="world"; else a+="!";

[language=JavaScript, title=(single-pass multi-execution)] var a="hello"; var env=navigator.plugins[0].name; // if branch a+="world"; // else branch a="hello"; a+="!";

GUI Event Analysis

In the DOM of most rich web applications, there are a variety of event handlers registered by different objects. For example, user can click on a button or submit a form. Event handler code may checks the state of GUI elements (e.g. check-box). User can trigger all those events in any order, and the application might have different behaviours. Some malicious content may only be triggered if victim triggers some events in certain order. This makes it very difficult to detect beforehand. [language=JavaScript, title=(examples of finding and triggering GUI elements in DOM)] // get all the DOM elements var allElements = document.getElementsByTagName('*');
// loop over all items and printout the one registered with onclick event for (var i = 0; i < allElements.length; i++) if (allElements[i].onclick) console.log(allElements[i]);
// check if a check-box is checked document.getElementById("myCheck").checked;
// trigger an onclick event document.getElementById("myButton").click(); Kudzu [**?**] develop a GUI explorer that searches the space of all event sequences using a random exploration strategy (randomly selects an ordering among the user events registered by the web page.)

One challenge is that event handler might be created or deleted during code execution. (i.e. after click the button, a new form object might be created or deleted in DOM). So if we could determine the priority of events, we can improve the efficiency of exploration. (in the previous example, the button that creates the new form should have higher priority then the form)

2.0.4 Statistics Models

χ^2 **algorithm:** This model is commonly used for testing correlation between features and malicious code. χ^2 test for one degree of freedom is described below:

A = malicious contexts with feature
B = benign contexts with feature
C = malicious contexts without feature
D = benign contexts without feature

$$\chi^2 = \frac{(A * D - C * B)^2}{(A + C) * (B + D) * (A + B) * (C + D)}$$

Bayesian statistics: This model had been used for classifier to classify malicious code and benign code based on training sets by assuming all features are independent. Even this assumption might not be true for example feature of string concatenation obfuscation might be related to feature of **eval** function call. However, surprisingly, this assumption has yielded good results in the past because of its simplicity which allows the classifier is efficient to train and run.

The probability assigned to label L_i for code fragment containing features $F_1, ..., F_n$ may be computed using Bayes rule as follows:

$$P(L_i|F_1, ..., F_n) = \frac{P(L_i)P(F_1, ..., F_n|L_i)}{P(F_1, ..., F_n)}$$

Chapter 3

PROJECT X

Chapter 4

Evaluation

Chapter 5

Conclusion

Appendix A

First Appendix