Automatic Detection for JavaScript Obfuscation Attacks in Web Pages through String Pattern Analysis

YoungHan Choi, TaeGhyoon Kim, SeokJin Choi
The Attached Institute of ETRI
{yhch,tgkim,choisj}@ensec.re.kr

Abstract

Recently, most of malicious web pages include obfuscated codes in order to circumvent the detection of signature-based detection systems .It is difficult to decide whether the sting is obfuscated because the shape of obfuscated strings are changed continuously. In this paper, we propose a novel methodology that can detect obfuscated strings in the malicious web pages. We extracted three metrics as rules for detecting obfuscated strings by analyzing patterns of normal and malicious JavaScript codes. They are N-gram*,* Entropy*, and* Word Size*.* N-gram *checks how many each byte code is used in strings.* Entropy *checks distributed of used byte codes.* Word size *checks whether there is used very long string. Based on the metrics, we implemented a practical tool for our methodology and evaluated it using read malicious web pages. The experiment results showed that our methodology can detect obfuscated strings in web pages effectively.*

Keywords: *JavaScript Obfuscation, Malicious Code Detection*

1. Introduction

JavaScript language has power that can execute dynamic work in the web browser. Therefore, malicious users attack client's system by inserting malicious JavaScript codes in a normal web page. Using JavaScript, they can steal personal information, download malware in client systems, and so on. In order to defend the attacks, security systems detect JavaScript codes in malicious web pages based on signatures. Nowadays, however, attackers circumvent the defense mechanism using obfuscation. Obfuscation is a method that changes shape of data in order to avoid pattern-matching detection. For instance, "*CLIENT ATTACK*" string can be changed into *"CL\x73\x69NT\x20\x65T\x84ACK"*. Because of obfuscation, many security systems recently fail to detect malicious JavaScript in web pages.

Again this background, we propose a novel methodology that detects automatically an obfuscated JavaScript code in a web page. After we analyze various malicious and normal web pages, we extract three metrics as rules for detecting obfuscation: *N-gram*, *Entropy*, and *Word Size*. *N-gram* is an algorithm for text search. We applied 1-gram to our detection algorithm. 1-gram is equal to byte occurrence frequency. Through entropy, we analyze the distribution of bytes. Because some obfuscated strings use often excessive long size, we define word size as third metric. Obfuscated strings are used in parameters of dangerous functions such as `eval` and `document.write`. Before detect obfuscated strings, we extract firstly all strings related to parameters of the functions. Using static data flow analysis, we trace data flow for variables in source codes of web pages. We apply three metrics into the strings and detect obfuscated strings. We implemented a practical tool for our methodology and experimented for real malicious web pages. The results showed that our methodology

detected obfuscated strings effectively. In this paper, we focus on JavaScript codes using obfuscation among various malicious web pages.

Our contribution is like this: **After we analyzed malicious web pages including obfuscated strings, we define three metrics as rules for detecting obfuscated strings. And then, we implemented a practical tool for our methodology and evaluated it.**

Our paper is organized as follows: In section 2, we introduce researches related to malicious JavaScript codes. Next, we propose our methodology for detecting automatically obfuscated strings in malicious JavaScript codes in section 3. In section 4, we classify JavaScript strings into three cases and propose a method that extracts all doubtful strings. In section 5, we defined three metrics for detecting obfuscated strings and evaluated it using real malicious web pages. Conclusions and direction for future work are presented in section 6.

2. Related Work

There are rich researches for detecting and analyzing malicious JavaScript codes in web pages. Because attackers use JavaScript in order to execute malicious work in a client system, many researches are performed for client defense.

In [1], authors studied various JavaScript redirection in spam pages and found that obfuscation techniques are prevalent among them. Feinstein analyzed JavaScript obfuscation cases and implemented obfuscation detection tool[3]. He hooked `eval` function and the string concatenation method based on Mozilla SpiderMonkey. This method has difficulty for modifying an engine of custom web browser. He found that use of the `eval` function was relatively more common in the benign scripts than in malicious scripts. In [8], the author introduced various malicious JavaScript attacks and obfuscation methods. He found that `eval` and `document.write` functions are mostly used in malicious web pages. These researches focus on malicious JavaScript codes itself. In [2], authors proposed the tool that can deobfuscate obfuscated strings by emulating a browser. They, however, focused on deobfuscation, but detection.

Provos *et al.* decided web pages as malicious pages if the pages caused the automatic installation of software without the user's knowledge or consent[10]. They found a malicious web page by monitoring behavior of Internet Explorer in a virtual machine dynamically. However, our method search obfuscation statically using source codes of web pages. In this research, they observed that a number of web pages in reputable sites are obfuscated and found that obfuscated JavaScript is not in itself a good indicator of malice. In [5], Ikinci implemented system for detecting malicious web pages. He, however, scanned only web pages using signature-based anti-virus program without considering obfuscation. Hallaraker *et al.* proposed a method that monitored JavaScript code execution in order to detect malicious code behavior and evaluated a mechanism to audit the execution of JavaScript code[4]. In [12], Wang *et al.* developed the tool that can detect malicious web pages using VM based on behavior of system, named HoneyMonkey.

In order to detect cross-site scripting, Vogt *et al.* tracked the flow of sensitive information insider the web browser using dynamic data tainting and static analysis[11]. Using static analysis, they traced every branch in the control flow by focusing on tainted value. In [6], authors proposed the method that can detect JavaScript worms based on propagation activity as worm's characters. They, however, didn't consider obfuscation of JavaScript. Wassermann

et al. presented a static analysis for finding XSS vulnerabilities that address weak or absent input validation[13]. They traced tainted information flows in web source codes.

3. Our Methodology for Detecting Obfuscation in Malicious Web Pages

In this chapter, we propose a novel methodology for detecting obfuscated strings in malicious web pages with Javascript codes. After we extract doubtful strings in web pages, we analyze them for deciding whether they are obfuscated or not. We extract all doubtful strings in web pages using static data flow analysis and detect obfuscated strings based on three metrics that we define. We named our algorithm the Javascript Obfuscation Detector in Web pages(JODW). In this paper, we focus on JavaScript codes using obfuscation among various malicious web pages.

JODW is a simple and strong method for detecting obfuscated string in malicious web pages. Firstly, it searches dangerous functions(`eval`, `document.write`, and so on) in web pages. The functions can execute strings of parameters. Because malicious user uses obfuscated strings in order to execute dynamic work after transmitting them as parameters of the functions, we start to parameters of the dangerous functions. Based on the parameters, we extract all strings related to them using static data flow analysis. After analyzing the strings, JODW detects obfuscated strings. Lastly, JOWD deofuscates the strings. In order to detect obfuscated strings in web pages, it demands many elements for automation.

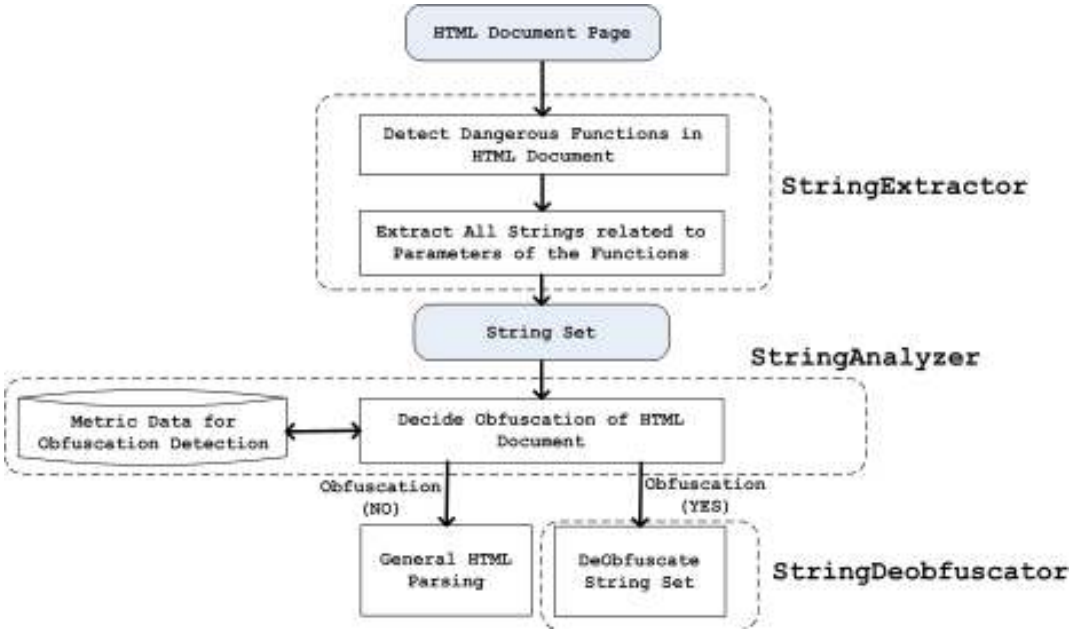


Figure 1. Our methodology for automatic detecting of obfuscated JavaScript strings in malicious web pages

Figure 1. shows our methodology and system for detection obfuscated strings in malicious web pages automatically. Our system makes up three modules: *StringExtractor*, *StringAnalyzer*, and *StringDeobfuscator*. It is as follows:

- **String Extractor** Most of malicious web pages call dangerous functions(`eval`, `document.write`) in order to perform malicious activity. Therefore, strings related to parameters of the dangerous functions have the high possibility that obfuscated strings are. We focus on the obfuscated strings. We trace all strings related to parameters of dangerous functions and extract all strings. We classify all strings of JavaScript into three cases. Based on the cases, we extract all strings using static data flow analysis. Static data flow analysis traces flow of data without executing web pages in a web browser. We will explain the mechanism of *StringExtractor* in chapter 4.
- **String Analyzer** This module decides whether previous doubtful strings are obfuscated or not. We define three metrics for detecting obfuscated strings in malicious web pages: *N-gram*, *Entropy*, *Word Size*. *StringAnalyzer* detects obfuscated string based on the three metrics. We will explain the method for detecting obfuscated codes in chapter 5.
- **String Deobfuscator** If obfuscated strings is detected, this module deobfuscates strings, and detects malicious codes in the deobfuscated string using patterns for malicious strings. For instance, in case that `IFRAME` is included in the strings, it is a malicious web page because a web browser parsing the tag connects a malicious web site automatically.

In this paper, we focus on *StringExtractor* and *StringAnalyzer* in order to detect obfuscated strings in malicious web pages. We will research for *StringDeobfuscator* that obfuscated string deobfuscates automatically in future.

4. Extraction of Obfuscated Strings in JavaScript

In this chapter, we explain the method that extracts all strings related to dangerous functions of JavaScript before performing the process for obfuscated string detection. Firstly, we classify all strings in JavaScript codes into three cases. Next, based on the three cases, we extract all strings related to parameters of dangerous functions using static data flow analysis.

4.1. JavaScript String Classification

In order to extract all strings related to parameters of dangerous functions, we classify all strings in JavaScript codes into three cases. Table 1 shows the definition for all strings in JavaScript. In the table, Y_l denotes a string of JavaScript codes. In order to check whether the string is obfuscated or not, we search all Y_l in malicious web pages. f function represent changes of strings and $x_1, x_2, x_3, ..., x_n$ are parameters.

Table 1. Definition for all strings in JavaScript

$$Y_l = \sum_{i=1}^{m} f_i(x_1, x_3, x_3, ..., x_n) \quad \begin{cases} Y_l \text{ is value of parameter} \\ x_1, x_3, x_3, ..., x_n, Y_l = string \\ n, m = 1,2,3,... \\ l = 1,2,3 \end{cases}$$

According to *f* and parameters, we classify all strings into three cases. In conclusion, all cases have the possibility of obfuscation. Therefore, all strings related to parameters of dangerous functions must be checked whether they are obfuscated or not. Three cases are as follows:

- **CASE1** : $Y_l = x_1$: **NoChange**

- **CASE2**: $Y_2 = \sum_{i=1}^{m} x_i$: **Concatenation**

- **CASE3** : $Y_3 = \sum_{i=1}^{m} f_i(x_1, x_2, ..., x_n)$: **Change**

CASE1(**NoChange**) : The function *f* is constant and has one variable, and *Y* is equal to x_1. Therefore, the variable x_1 is directly transmitted to a parameter of `eval` or `document.write`. For instance, an example code is as follows: `var x1 = "3+4"; ...; eval(x1);` `eval` function uses the string allocated in x_1 without modifying the value. However, another example shows a obfuscated string as follows: `eval("\144\157\143\165\155\145\156\164")`. This case is executed after the string is decoded. Therefore, we analyze the string whether it is decoded or not. In static data flow analysis, this case is extracted and analyzed directly.

CASE2(**Concatenation**) : The function *f* is constant. However, *Y* concatenates several strings such as $x_1, x_2, x_3, ..., x_n$. Because this case also has the possibility that *Y* is obfuscated, analysis for obfuscation detection must be performed. For instance, an example code is as follows:

```
var x1 = "te ActiveX Co"; var x2 = "ntrol"; var x3 = x1 + x2; var
x4 = "Execu" + x3; ... eval(x4);
```

In this example, x_4 is "`Execute ActiveX Control`". However, because the string is divided into several strings, a signature-based detect system can't detect the string. Therefore, the case must be analyzed for obfuscation detection. In static data flow analysis, we divide the string by a plus(+) character, save each strings, and analyze for obfuscaion detection.

CASE3(**Change**) : Various functions in malicious web pages decode obfuscated strings. The functions can be JavaScript functions or user-made functions. In static data flow analysis, we trace the functions and extract strings related to parameters of them. For instance, an example code is as follows:

```
Uul1ItLo["plunger"] = new Array(); var Qn2_R5kv = new Array(32,64,
256, 32768); for (var auLRkELh = 0; auLRkELh < 6; auLRkELh++)
{ for(var x8n9EKml = 0; x8n9EKml < 4; x8n9EKml++) { var CRrtOhOH =
Uul1ItLo["plunger"].length;  eval('Uul1ItLo["plunger"][CRrtOhOH]  =
GjL08iWK.substr(0, ('+ Qn2_R5kv[x8n9EKml] + '-6)/2);'); } }
```

In this example, *Y* is *Uul1ItLo["plunger"][CRrtOhOH] = GjL08iWK.substr(0, (Qn2_R5kv[x8n9EKml]-6)/2);*. f_1 is *Uul1ItLo["plunger"][CRrtOhOH],* f_2 is =, and f_3 is *GjL08iWK.substr(0, (Qn2_R5kv[x8n9EKml]-6)/2).* f_1 and f_3 are functions, and f_2 is constant. Therefore, f_1 and f_3 are changed by the decoding functions.

4.2. Extraction of Obfuscated String using Static Data Flow Analysis

Based on three cases for strings of JavaScript, we search and extract all strings related to dangerous functions using static data flow analysis. Static data flow analysis is to trace data flow of variables in source codes without executing a program. We trace all strings dangerous functions written JavaScript in web pages. We focus on `eval` and `document.write` as dangerous functions because the function is used in most of JavaScript obfuscation.

Table 2: Our algorithm for extracting all strings in HTML web pages using static data flow analysis

−**INPUT**: HTML Web page(*W*)
−**OUTPUT**: All strings related to dangerous functions(S{ $s_i, i = 0,1,...,n$)
Extract all JavaScript codes in *W*
Search dangerous functions and their function pointer reassignment
Update function list(*PL*{ $pl_i, i = 0,1,...,n$ })
A1 :
Trace pl_i
Analyze parameters of the functions
Classify the parameters into strings and functions(P{ $pl_i, i = 0,1,...,n$ })
A2 :
Check what is p_i 's case
    If p_i is string(*CASE1*), pi is saved
    If p_i is string concatenation(*CASE2*), divide p_i ,update p_i ,and GOTO **A2**
    If p_i is function(*CASE3*), update *PL* and GOTO **A1**
Extract all strings(*S*) related to dangerous functions

We name the methodology the static data flow analysis in JavaScript(SDFAJ). SDFAJ extracts all strings in web pages from parameters of dangerous function reversely. The algorithm for SDFAJ is shown in Table 2. SDFAJ firstly scans text in a web page and extracts all JavaScript codes in `<script>` and `</script>`. Next, SDFAJ search dangerous functions and their function pointer reassignment. In order to hide the call of dangerous functions in a malicious web page, a malicious user uses function pointer reassignment such as `function1 = eval`. SDFAJ traces variables related the functions, and divides them into simple strings and functions. Considering three string cases, SDFAJ traces all strings from the dangerous functions. Using the strings, SDFAJ checks what case it is(**A2**). If it is a string, SDFAJ saves the value because it traces the string no more. In case of CASE2, SDFAJ splits strings by a plus(+) character, and saves each value. If it is a function, SDFAJ analyzes parameters, and divides them into simple strings and function recursively(**A1**). By doing this, SDFAJ extracts all strings related to parameters of dangerous functions in malicious web pages.

An example that SDFAJ extracts all strings related to parameters of `eval` is shown in Figure 2. The value of parameter of `eval` is `ab3+ab2+"cccd"`. SDFAF divides it into `ab3`, `ab2`, and `"cccd"` by `+` character. Because `"cccd"` is CASE1, `cccd` is saved as a string. Because `ab2` is CASE3, it search parameters of the function and extracts the parameter of it.

It is `112, 108, 97, 105, 110` and is extracted as a string. Lastly, `ab3` is CASE2 and is divided into `"ddd"` and `ab1`. `ddd` is saved because it is a simple string. After SDFAF traces `ab1`, it saves `cccc` in string list. In the example, SDFAJ extracts four strings, such as `"ccc"`, `"112, 108, 97, 105, 110"`, `"ddd"`, and `"cccd"`. Using these strings, SDFAJ decides whether they are obfuscated.

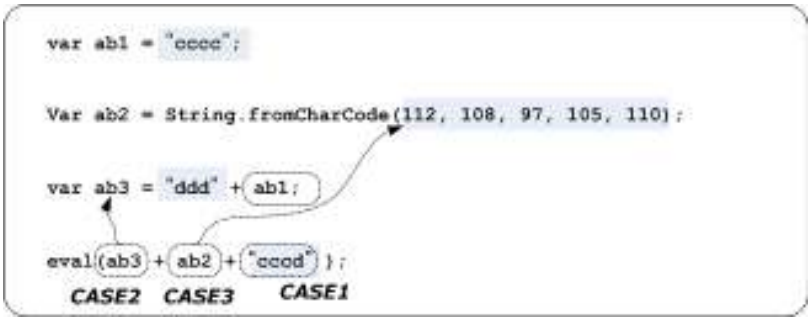


Figure 2. An example of string extracting using static data flow analysis in JavaScript code

5. Detection of Obfuscated Strings in JavaScript Codes

In this chapter, we propose the method to detect JavaScript obfuscated strings in a web page using all strings extracted by static data flow analysis. In order to check whether a string is obfuscated or not, we define three metrics: *N-gram*, *Entropy*, and *Word Size*. N-gram is an algorithm for text search. We search patterns for detecting obfuscated strings after analyzing normal and malicious web pages. Based on the metrics, we made experiments on detection for obfuscated strings in real malicious web pages.

5.1. Metrics for Detecting Obfuscated Strings

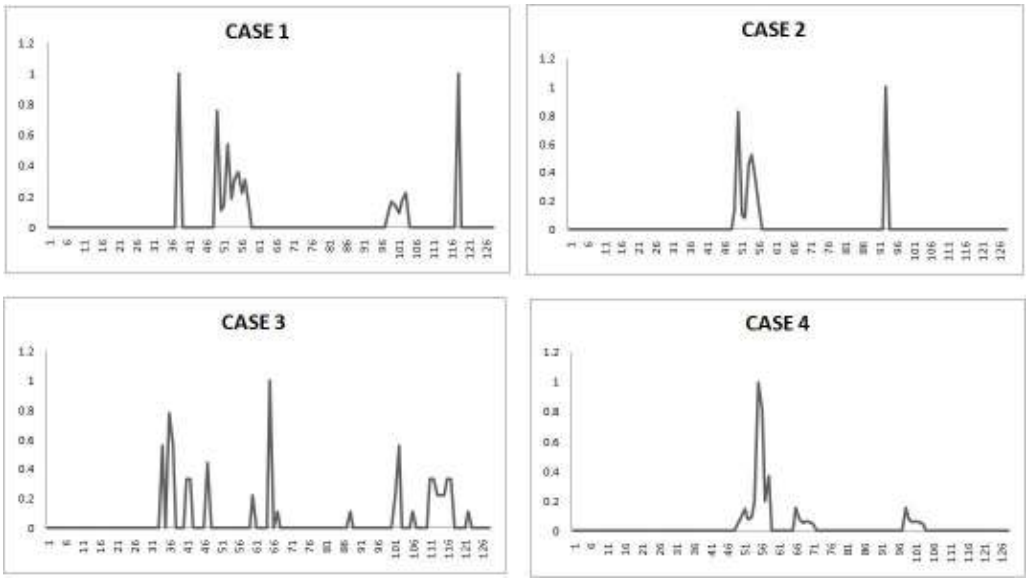
We define three metrics for detecting obfuscated strings. We search the usage frequency of ascii code in the strings using byte occurrence frequency as 1-gram. In order to know distribution of characters in strings, we calculate the entropy based on 1-gram. Lastly, we analyze the size of word because most of obfuscated strings are very long.

- **N-garm** checks how many each byte code is used in strings
- **Entropy** checks distribution of used byte codes
- **Word Size** checks whether there is used very long string

In order to analyze patterns of normal web pages, we downloaded web page files including various JavaScript codes. Using *OpenWebSpider*[9], we collected web pages. *OpenWebSpider* is an open source web crawler and controls information about web pages in *MySQL*[7] database. Using *OpenWebSpider# v0.1.3*, we collected web page files in sub directories of 100 web sites and analyzed them.

5.1.1. N-gram: We check the usage frequency of ascii code in the strings. By doing this, we can know how many each byte code is used in the string. We use 1-gram among N-gram and it is equal to byte occurrence frequency. We classify ascii code into three category as

shown in Table 3. We focus on *Special Char* among byte codes, because much obfuscated strings use excessively specific characters such as `\`, `[`, `]`, `@`, `x`, `u`, and so on. In strings of



normal JavaScript codes, *Alphabet* and *Number* are used evenly among all.

Figure 3. Byte occurrence frequency for various JavaScript Obfuscated strings. X axis is ascii code number. Y axis is total number of each byte used in strings and we normalize values of Y axis by maximum value. The target of CASE1 is the string such as "%u9495%u4590...'', and CASE2 is "\144\156''. CASE3 is the case that obfuscated string uses various characters, and CASE4 uses *Alphabet* and *Number*.

Table 3. Ascii Code

Name	*Ascii Code Number*	*Character*
Alphabet	0x41-0x5A, 0x61-0x7A	A-Z, a-z
Number	0x30-0x39	0-9
Special Char	0x21-0x2F 0x3A-0x40 0x5B-0x5F, 0x7B-0x7E	! " # $ % & ' { } * + , - . / : ; < = > ? @ [\ ] ^ _ ` { | } ~

We analyze various obfuscated strings in malicious web pages. Figure. 3 shows patterns of some obfuscated strings. In this chapter, we analyze four cases among various patterns of the strings. There exists on various patterns except for the cases. The strings of CASE1 and CASE2 are related to the decoding mechanism that JavaScript language offers. The stings such as "%u9495%u4590..." and "\144\156" are decoded in `eval` function directly, or in `escape` and `unescape` functions. They have excessively specific characters such as "`%u`", "\", and so on. CASE3 and CASE4 are examples that obfuscated strings need user-made decoding functions. In cases, there are various patterns of byte occurrence frequency according to decoding functions. CASE3 has characters in all extent. CASE4 has *Alphabet*

and *Number* characters intensively. *We decide that a string is obfuscated if it uses some specific characters excessively*. Based on the analysis result, we define metric1 as follows:

Metric1 : **Byte Occurrence Frequency of Specific Character** *Obfuscated string uses some specific characters in the string excessively*

5.1.2. Entropy: In order to analyze the distribution of bytes, we define entropy as second metric. Entropy is calculated as follows:

$$E(B) = -\sum_{i=1}^{N} (\frac{b_i}{T}) \log(\frac{b_i}{T}) \quad \begin{cases} B = \{b_i, i = 0,1,..., N\} \\ T = \sum_{i=1}^{N} b_i \end{cases}$$

In entropy(E), b_i is count of each byte values and T is total count of bytes in a string. If there are some bytes in a string, E reaches zero. Maximum value of E is $logN$ and it means that byte codes are distributed widely throughout whole bytes. N is 128 because we focus on readable strings.

Figure 4. Entropy for various JavaScript strings. Y axis is value of entropy. Upper region represents that the string has a whole characters widely. Middle region represents entropy for general JavaScript codes and sentences. Lower region is that the string has some specific characters excessively and has possibility of obfuscated string.

We calculate entropy of previous four cases. CASE1 is 1.12249, CASE2 0.82906, CASE3 1.24406, and CASE4 1.09014. The string that includes some kinds of characters has a low value of entropy, and vice verse. In search entropy ranges of ascii code, we calculate entropy of a string including all ascii code. The entropy is 1.97313. We select this as maximum value. An entropy of JavaScript codes in a general web page is roughly 1.6496. Because most strings are readable sentences, we calculate the entropy of sentences. The strings use alphabet, number and some special characters(, . “ ”). The entropy is about 1.3093. Collectively, range of entropy for various JavaScript strings is shown in Figure 4. We set up two ranges: one region for some specific character, and the other region for general JavaScript codes and sentences. In Figure 4, we exclude the case because entropy of upper region is an ideal case.

We decide that a string is obfuscated if entropy is less than 1.2. Based on the analysis result, we define metric2 as follows:

Metric2 : **Entropy** *obfuscated string has the low value of entropy because it uses some characters*

5.1.3. Word Size: We define the word size as third metric. Because words in JavaScript code are generally read by man, their sizes are not so long. Many obfuscated strings use very long word size. For instance, the word size in a malicious web page that we collect is 9,212 bytes. In this metric, we target on a normal string. We divide stings into words by a space character(`0x20`). In order to analyze range of normal word size, we analyze various sentences and JavaScript codes as shown in Fig. 5. In the figure, (a) represents the distribution of word size in a general sentence. In the sentences, word size is under 30 on the average. (b) is the distribution of word size in a JavaScript code. The range of word size is from 0 to 300 generally. Based on the analysis result, we define metric3 as follows:

Metric3 : **Word Size** Obfuscated string has excessive long size of word

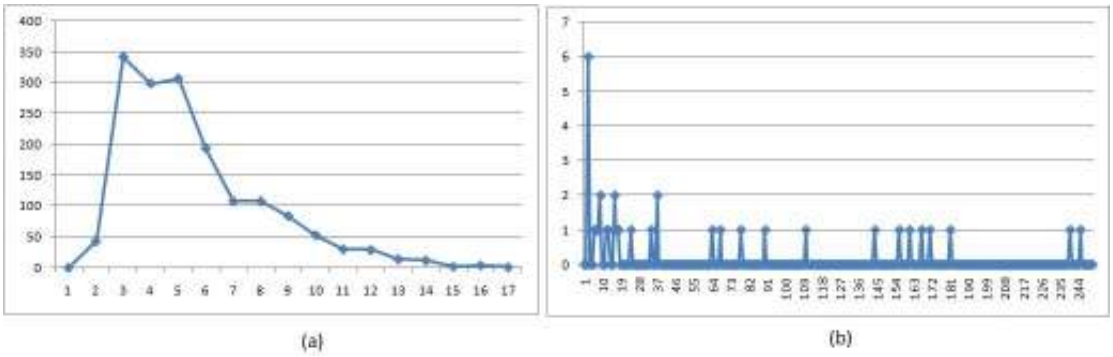


Figure 5. Word size in general sentences and JavaScript codes. (a) is the distribution of word size in general sentences. (b) is the distribution of word size in a JavaScript code

5.2. Evaluation and Experiments

In this chapter, we applied our three metrics into malicious web pages with obfuscated strings. We collected 33 real malicious pages. Among these pages, 14 pages include obfuscated strings. However, some pages are analogous to each other. We exclude similar pages and experiment 6 kind of malicious web pages. Therefore, we applied our methodology to 6 patterns of obfuscated strings in malicious web pages. Table 4 shows patterns of each obfuscated string. We set up value of our metrics as follows:

- **Metric1** If the string includes *Special Char* excessively, it is obfuscated.
- **Metric2** If entry of the string is less than 1.2, it is obfuscated.
- **Metric3** If word size is more than 350, it is obfuscated.

Results of detection for obfuscated strings in each file based on three metrics are shown in Table 5.

- **File1 is Obfuscated String** Word size is 750 bytes and entropy is less than 1.2. It includes a backslash(92) character excessively.
- **File2 is Obfuscated String** Word size is 531 bytes. It include excessively a special character(124) that does not used mostly in general strings.
- **File3 is Obfuscated String** Word size is 9,212 bytes. It is very long size.
- **File4 isn't Obfuscated String** Entropy is more than 1.2, and maximum word size is 29. Metrics for this pattern is false alarm.
- **File5 is suspicious of Obfuscated String** It includes bytes code evenly among all and entropy is more than 1.2. However, it has long word size more than 350 bytes.
- **File6 is Obfuscated String** It includes some special characters(33 35 36 64).

Among total 6 malicious patterns, we found obfuscated strings in 4 patterns of malicious web pages and suspected that one pattern includes an obfuscated string. However, we cannot find one malicious pattern.

Table 4. Partial obfuscated strings in each malicious web page. *File1* and *File6* is the string decoded by JavaScript functions. *File2* is a JavaScript Code. *File3*, *File4*, and *File5* is the string decoded by user-made decode functions

File Name	*Obfuscated String*
File1	*\144\157\143\165\155\145\156\164\56*
File2	*||var|document|object|expires|if|finally|catch|write||*
File3	*97ACA29baca2B3A5517A99696Bae9B677d995C876a7*
File4	*eval(rmdiyfrT+eSS9YDtk[VFtUaNvX]);*
File5	*a1443oe.setTime(a1443oe.getTime()+365*24*60*60*1000);*
File6	*t!.Wr@i@te(#$q!.res$po#n#s$e@Bo@dy@)@'.replace(/\!|@|#|\$/ig, '')*

Table 5. Results of detection for obfuscated strings. Metric1 is byte occurrence frequency, Metric2 is entropy, and Metric3 is the longest word size in the file. Values of Metric1 are byte codes used over 50% of maximum frequency number

FileName	*Metirc1*	*Metric2*	*Metric3*	*Detection*
File1	49 53 92	0.82906	750	YES
File2	40 41 49 50 101 116 124	1.71222	531	YES
File3	54 55 57	1.15289	9212	YES
File4	34 101 116	1.61563	29	NO
File5	49 51 52 59 61 97 101 111 112 114 116	1.65364	364	SUSPICIOUS
File6	33 35 36 64 101	1.4207	87	YES

6. Conclusion and Future Work

In this paper, we proposed a novel methodology that can detect obfuscated strings in malicious web pages. Recently, Obfuscation is used by malicious attackers in order to circumvent the detection of signature-based security systems. After we analyzed patterns of malicious and general web pages, we defined three metrics as obfuscation detection rules: N-gram, Entropy, and Word Size. Based on the metrics, we applied our methodology into real malicious web pages and evaluated them. The results show that the methodology found 4 patterns for obfuscated strings among 6 patterns for real malicious web pages. It means that

our methodology is effective for detecting obfuscated strings. We, however, have limitations that number of patterns our methodology can detect is few, because we has a few patterns of real malicious web page. As future work, we will be centered on finding new metrics for detecting obfuscated strings. And we will research the methodology that can deobfuscate obfuscated strings automatically.

References

[1] Chellapilla, K., Maykov, A.: A Taxonomy of JavaScript Redirection Spam. In: Proceedings of the 3rd International Workshop on Adversarial Information Retrieval on Web (AIRWeb 2007) (2007)

[2] Chenetee, S., Rice, A.: Spiffy: Automated JavaScript Deobfuscation. In: PacSec 2007 (2007)

[3] Feinstein, B., Peck, D.: Caffeine Monkey: Automated Collection, Detection and Analysis of Malicious JavaScript. Black Hat USA (2007)

[4] Hallaraker, O., Vigna, G.: Detecting Malicious JavaScript Code in Mozilla. In: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECC 2005) (2005)

[5] Ikinci, A., Holz, T., Freiling, F.: Monkey-Spider: Detecting Malicious Websites with Low-Interaction Honeyclients. In: Proceedings of Sicherheit 2008 (2008)

[6] Livshits, B., Cui, W.: Spectator: Detection and Containment of JavaScript Worms. In: Proceedings of the USENIX 2008 Annual Technical Conference on Annual Technical Conference (2008)

[7] MySQL - open source database, http://www.mysql.com

[8] Nazario, J.: Reverse Engineering Malicious Javascript. In: CanSecWest 2007 (2007)

[9] OpenWebSpider - open source web spider, http://www.openwebspider.org

[10] Provos, N., McNamee, D., Mavrommatis, P., Wang, K., Modadugu, N.: The Ghost in the Browser Analysis of Web-based Malware. In: First Workshop on Hot Topics in Understanding Botnets (2007)

[11] Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C., Vigna, G.: Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In: Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS 2007) (2007)

[12] Wang, Y., Beck, D., Jiang, X., Roussev, R., Verbowski, C., Chen, S., King, S.: Automated Web Petrol with Strider HoneyMonkey. In: Proceedings of the Network and Distributed System Security Symposium (NDSS 2006) (2006)

[13] Wassermann, G., Su, Z.: Static Detection of Cross-Site Scripting Vulnerabilities. In: Proceedings of the 30th International Conference Software Engineering (ICSE 2008) (2008)

Authors

Young Han Choi is currently a senior research engineer in the Attached Institute of Electronics and Telecommunications Research. His research interests include software security, intrusion detection systems, and operating system. He received his B.Sc. and M.Sc. degrees in electronic engineering from Hanyang University and Korea Advanced Institute of Science and Technology, Korea, in 2002 and 2004, respectively.

Tae Ghyoon Kim is currently a senior research engineer in the Attached Institute of Electronics and Telecommunications Research. His research interests include software security, intrusion detection systems, and operating system. He received his B.Sc. and M.Sc. degrees in electronic engineering from Chungnam National University, Korea, in 1995 and 1997, respectively.

Seok Jin Choi is currently a senior research engineer in the Attached Institute of Electronics and Telecommunications Research. His research interests include software

security, intrusion detection systems, and operating system. He received his B.Sc. and M.Sc. degrees in electronic engineering from Kyungbook University and Korea Advanced Institute of Science and Technology, Korea, in 1995 and 1998, respectively.