Detection of Heap-Spraying Attacks Using String Trace Graph

Jaehyeok Song, Jonghyuk Song, and Jong Kim(✉)

Department of CSE, POSTECH, Pohang, Republic of Korea
{the13,freestar,jkim}@postech.ac.kr

Abstract. Heap-spraying is an attack technique that exploits memory corruptions in web browsers. A realtime detection of heap-spraying is difficult because of dynamic nature of JavaScript and monitoring overheads. In this paper, we propose a runtime detector of heap-spraying attacks in web browsers. We build a *string trace graph* by tracing all string objects and string operations in JavaScript. The graph is used for detecting abnormal behaviors of JavaScript. We detect heap-spraying attacks with low false positive rate and overheads.

1 Introduction

In recent years, a drive-by-download attack becomes one of the most common methods to spread malware. Attackers tempt a victim to visit a website that contains a malicious code. The malicious code exploits vulnerabilities of a web browser to compromise a victim's computer. Compromised computers are used as components of botnets and conduct various attacks, such as spamming and distributed denial-of-service attack (DDoS). Various techniques are used in order to load shellcode into the memory and execute it.

Heap-spraying is the most common technique to compromise web browsers. Heap-spraying increases the possibility of successful attacks because attackers do not need to know exact heap addresses. Heap-spraying is carried out in two phases. The first phase is building a code block that contains a large chunk of CPU instructions. The code block consists of two parts: NOP-sled and shellcode. NOP-sled contains meaningless CPU instructions that induce execution to a malicious shellcode. In the second phase, the malicious code makes many copies of the code block. Heap-spraying tries to insert the code block as many as possible to increase the possibility of the attack. Therefore, heap-spraying technique uses a large amount of memory. In the real world, malicious JavaScript that uses heap-spraying usually allocates more than 100 MB of memory. In addition, heap-spraying should use only string objects of JavaScript to build the code block. The string object is the only object that controls each byte of memory, so heap-spraying uses JavaScript string objects.

In this paper, we propose a heap-spraying detection method based on a string trace graph. Our method builds a graph by tracing all string operations in JavaScript. We propose three features from a string trace graph and train

K.-H. Rhee and J.H. Yi (Eds.): WISA 2014, LNCS 8909, pp. 17–26, 2015.
DOI: 10.1007/978-3-319-15087-1_2

classifiers using the features to classify heap-spraying codes. We evaluate our method by using real-world data and evaluation results show that our method has low false positive and overheads.

We organize the remainder of this paper as follows. In Sect. 2, we introduce previous heap-spraying detection methods and malicious JavaScript detection methods. Section 3 explains background knowledge to understand our method. In Sect. 4, we explain our detection method in detail. In Sect. 5, we describe evaluation results. Finally, Sect. 6 concludes the paper and presents future work.

2 Related Work

2.1 Heap-Spraying Detection

Previous studies [6,13,14] have proposed to detect heap-spraying by finding sequences of x86 instructions. The heap blocks used in a typical heap-spraying attack contain a shellcode and the remainder of the heap block contains NOP-sleds. Previous studies focus on identifying large chunks of NOP-sleds. Nozzle [14] disassembles given a heap object with possible x86 instructions by building a control flow graph (CFG). However, Ding *et al.* [9] proved that Nozzle is broken by manipulating heap behaviors. Nozzle has too high overhead because it scans the contents of all allocated heap memory. We propose a method that has lower overhead than Nozzle because we do not check the contents of the memory. We simply check whether a memory is allocated and also get the size of allocated memory.

Several researches [6,12,16] have proposed to detect executable codes in payloads of network packets, but they have high false positives.

2.2 Malicious JavaScript Detection

There are researches to detect other malicious JavaScript codes, such as obfuscation, exploit or fingerprint. A number of server-side approaches [7,10] have been proposed to identify malicious code on the web. These approaches extract features of each webpage in run-time using an emulated browser. Cova *et al.* [7] propose a method to detect malicious JavaScript codes, but it takes too much time to analyze each page (about 10 s per page). In addition, the server-side approaches always suffer from IP based filtering and also have a lot of false positives.

A proxy approach [15] also has been proposed. It detects obfuscation and exploit code by dynamic and static analysis in the emulated environment at proxy level.

Zozzle [8] uses nearly-static approach to detect malicious JavaScript. When a JavaScript engine evaluates a source code, Zozzle analyzes the source code statically. Similar to our method, Zozzle uses a machine learning technique to classify malicious JavaScript. Zozzle trains the name of variables and functions as a feature but an attacker simply changes a variable name in source code or uses a JavaScript optimization compiler [2] to avoid Zozzle.

3 Background

3.1 A String Object in JavaScript

In JavaScript, a string object has unique characteristics distinguished from other languages. First, a value of string is immutable. This means that once a string is initialized, the value of the string will not be changed. Every string operation create a new string variable instead of modifying the original value [10].

```
<SCRIPT language="text/javascript">
 1   var shellcode = unescape("%u4343%u4343%...");
 2   var nopsled = unescape("%u0D0D%u0D0D");
 3
 4   while (nopsled.length<0x40000) {
 5      nopsled += nopsled;
 6   }
 7   var nopsled = nopsled.substring(0,0x40000 - shellcode.length);
 8
 9   spray = new Array();
10   for (i=0; i<200; i++) {
11      spray[i] = nopsled + shellcode;
12   }
</SCRIPT>
```

Fig. 1. An example code of a typical heap-spraying in JavaScript

Second, a string is the only object to manipulate a memory in JavaScript. To succeed code injection attack, an attacker has to load a malicious code on the memory. Since the code consists of a sequence of CPU instructions, each byte of the code has to be accessible. In JavaScript, a string object is the only candidate to have that functionality among user controllable objects.

From the above characteristics, we can know that attackers exploit string objects in JavaScript to manipulate the memory.

3.2 Heap-Spraying

There are two types in code injection attacks which are stack-based and heap-based. Stack-based attacks are on the decline because numerous methods have been introduced to prevent the stack-based attacks. Therefore, attackers mainly

```
<SCRIPT language="text/javascript">
 1  var x = "foo";
 2  var y = "bar";
 3  var z = "hello world";
 4
 5  var a = x + y + " " + z;
 6  // a has "foobar hello world"
 7
 8  for(var i=0;i<5;i++) document.write(a.substring(0,5));
 9  // "foobafoobafoobafoobafooba" is printed on a page
</SCRIPT>
```

Fig. 2. An example code to explain the string trace graph

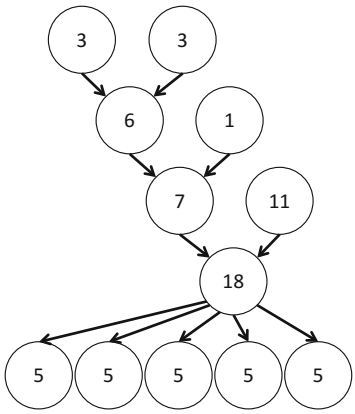


Fig. 3. An example graph of string trace graph

use heap-based attack to compromise victims. Heap-based attack is more difficult than stack-based attack because the addresses in heap memory are unpredictable. To overcome this trouble, attackers should adopt several strategies such as heap-spraying.

Figure 1 is an example code of a heap-spraying in JavaScript. Lines 1–2 indicates that allocating shellcode and NOP-sled into strings. Lines 4–7 build NOP-sleds to spray. In the first while loop, the NOP-sled is expanded by concatenating itself. When the NOP-sled is expanded, the NOP-sled is sliced to fit into the size of heap memory chunk. Although the NOP-sled size is different from a target of browsers and platforms, it has large size of memory than the memory page size, typically from 128 kB to 524 kB. Lines 9–12 codes are responsible for combining the NOP-sled with the shellcode. In this step, the code makes many copies for the effectiveness of the attack.

In our observation of heap-spraying analysis, we found out three features of heap-spraying. First, NOP-sled is generated from the small number of short strings because an exploit should be performed in a short time. If the exploit takes long time, a victim stops navigating the site.

Second, heap-spraying uses abnormally long strings. Attackers insert NOP-sleds as much as possible to increase the possibility that a jump instruction lands on the NOP sled. If a jump instruction lands on the NOP-sled, the execution is reached to a shellcode. Therefore, heap-spraying needs a large size of the NOP-sled string to increase effectiveness of the attack.

Third, heap-spraying makes many copies of a block that contains NOP-sleds and a shellcode. Increasing the number of the block that contains the attack codes is another way to increase the probability of the attack. Therefore, the block is copied the hundreds of times.

4 Heap-Spraying Detection Based on a String Trace Graph

We detect heap-spraying based on a string trace graph. We trace all string operations in JavaScript, such as concatenation, replacement and substring. Therefore, we can get an execution history of string operations by generating a string trace graph.

A string trace graph $\mathcal{G}$ consists of nodes $\mathcal{V}$ and directed edges $\mathcal{E}$. Each node $\mathcal{V}$ represents a string object and it has a length of the string as an attribute. There are two node types which are a leaf node $\mathcal{V}_{Leaf}$ and an internal node $\mathcal{V}_{Internal}$. A leaf node $\mathcal{V}_{Leaf}$ has no incoming edges but an internal node $\mathcal{V}_{Internal}$ has incoming edges. Initial strings are represented as leaf nodes and output strings of string operations are represented as internal nodes. Directed edges represent execution flows of string operations.

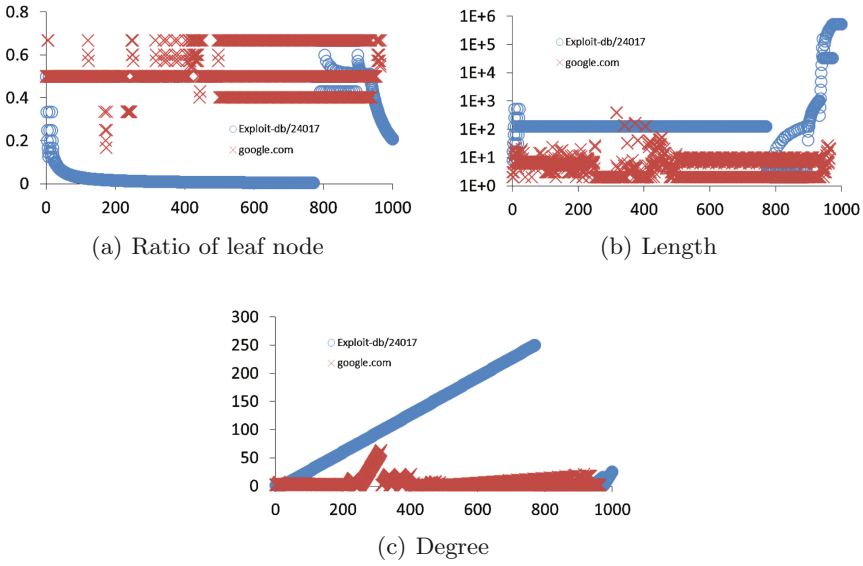
Figures 2 and 3 show how we create graphs from JavaScript codes. Each node represents a string object and the number means the length of the string. Each edge represents a flow of a string operation. By analyzing the graph in Fig. 3, we can know that there are four initial strings and a string operation is performed repeatedly in the last part of the graph.

4.1 Features

We propose three features to detect heap-spraying attacks: ratio of leaf nodes, length of a string and degree of the nodes. First, our method uses a ratio of leaf nodes as a feature to detect heap-spraying. A ratio of leaf nodes $LeafR_{\mathcal{G}}$ of a string trace graph $\mathcal{G}$ is computed as follows.

$$LeafR_{\mathcal{G}} = \frac{n_{leaf_{\mathcal{G}}}}{(n_{leaf_{\mathcal{G}}} + n_{internal_{\mathcal{G}}})}, \tag{1}$$

where $n_{leaf_{\mathcal{G}}}$ is the number of leaf nodes in $\mathcal{G}$ and $n_{internal_{\mathcal{G}}}$ is the number of internal nodes in $\mathcal{G}$. Heap-spraying has a few leaf nodes because it begins from



(a) Ratio of leaf node

(b) Length

(c) Degree

Fig. 4. Comparison of the features for google.com and a published exploit 24017 in exploit-db.com. x axis for the number of generated strings and y axis for each feature.

a small number of initial strings. In general, JavaScript codes in the normal websites have a lot of initial strings to represent texts.

Second, our method uses the length of strings to detect heap-spraying. Heap-spraying exploits long strings because it makes many NOP-sleds to increase possibility of the attack. Each node of the string trace graph contains the length of string objects, so we can detect abnormally long strings.

Third features is the degree of a node that is the number of outgoing edges of the node. The degree of a node represents how many string operations are performed with the node. Heap-spraying performs string operations many times to copy an object that contains NOP-sleds and a shellcode to increase the possibility of attacks. If string operations are performed many times, there is a node having an unusually larger number of outgoing edges. If there is a nodes that has a larger degree than a threshold, our method decides that there is a heap-spraying attack.

We train well-known classification algorithms with these three features. The trained classification algorithms classify whether a JavaScript contains heap-spraying codes.

5 Evaluation

We implement our method on JavaScriptCore (JSC) which is a default JavaScript engine of an open-source web engine Webkit [5]. The release version that we modify is r128399. We modify JavaScript String class to trace every constructor and destructor. Our code is written in 600 lines of code.

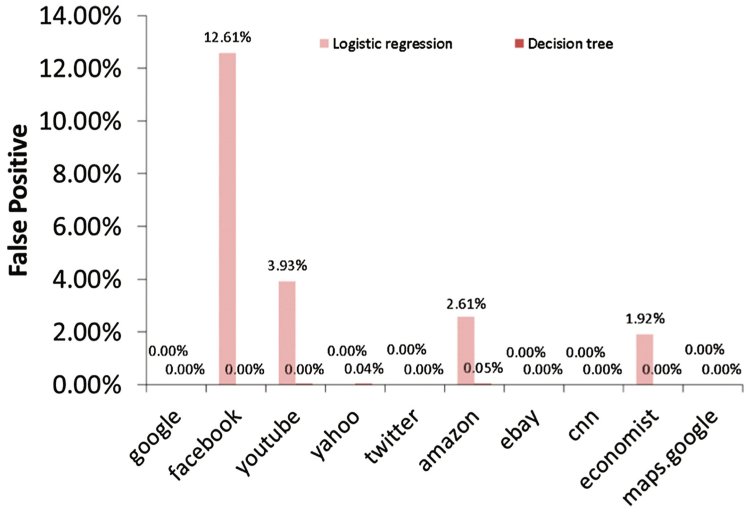


Fig. 5. Comparison of false positive rate for 10 benign web sites

We begin our evaluation by measuring the effects of each three feature mentioned in Sect. 4.1. Figure 4 shows the results on a benign website (google.com) and a site that contains a published heap-spraying attack code. Figure 4(a) shows the ratio of leaf nodes. The ratio of leaf nodes in the heap-spraying is much lower than that of the benign. In general, normal websites contain many initial strings to represent text but a heap-spraying code only uses a few initial strings for setting up attack blocks. Figure 4(b) shows the result of the length feature. The length of strings in the heap-spraying is much longer than that of benign because the heap-spraying uses abnormally long strings to increase the possibility of the attack. Figure 4(c) shows the result of the degree feature. The maximum degree of benign is 80 but the degree of heap-spraying is much higher than that. From this result, we can know that the heap-spraying code performs string operations much more than a benign website.

Overall, three features are very useful to distinguish between a malicious site that contains a heap-spraying code and a benign site.

5.1 False Positive Rate and False Negative Rate

In this section, we compute false positive rate and false negative rate. We crawl the front pages of Alexa top 500 sites [1] as a benign data set. We set up a malicious data set with 50 web sites in malwaredomainlist.com [4] and a published heap-spraying sample in exploit-db.com [3].

Weka [11] is used for classifications. We use 66 % of our data set for training and the remainder is used for validation. Four classifiers are trained: decision tree, logistic regression, naive Bayes and SVM. Table 1 shows the results of the four

Table 1. False positive and false negative of classifiers trained by four algorithms; decision tree, logistic regression, Naive Bayes and SVM.

Algorithms	False positive rate (%)	False negative rate (%)
Decision tree	0.00	0.00
Logistic regression	0.03	0.98
Naive Bayes	4.00	0.67
SVM	18.50	0.00

Table 2. Benign web sites that we used in experiments

Sites	Document (kB)	JavaScript (kB)
google.com	98	787
facebook.com	80	389
youtube.com	99	352
yahoo.com	316	651
twitter.com	52	306
amazon.com	227	316
ebay.com	75	272
cnn.com	110	1232
economist.com	150	610
maps.google.com	205	797

classifiers. Decision tree and logistic regression classifiers achieve outstanding performance. The others also have low false positive rate and false negative rate.

To examine the results of false positive rate in detail, we select the results of 10 popular sites (Table 2) that are classified with two classifiers which are decision tree and logistic regression. We visit not only the front page but also up to 20 internal pages of the sites. Figure 5 shows the result. Overall, logistic regression performs with low false positive rates and decision tree performs with almost zero false positive rates. In facebook.com case, logistic regression has the highest false positive rate because facebook.com has many copy operations. False positives are mainly caused from some benign sites implemented with obfuscated codes or many string operations.

5.2 Performance

We evaluate the time overhead and the memory usage on the 10 sites (Table 2). When we measure the performance, we except network and rendering overheads by executing only JSC with Webkit. The total memory usage is calculated by counting the number of objects. We conduct the evaluation with decision tree and logistic regression because these two algorithms have the low false positive rate and false negative in Sect. 5.1.

Fig. 6. Run-time overhead results

Fig. 7. Memory usage overhead results

Figure 6 shows the results of run-time overhead. Decision tree takes more time about 11 % and logistic regression takes more time about 12 %. Figure 7 shows the results of memory usages. On average, our method uses approximately 2.3 % additional memory. In facebook.com and twitter.com, we consume more time and memory because they have more string operations than other sites. Google Map site uses a lot of memory, so the additional memory usage is relatively too small for the existing memory usage.

6 Conclusions and Future Work

This paper proposed a heap-spraying detection method based on a string trace graph. We build a graph by tracing all string operations in JavaScript. Our method is executed in a client browser and checks every web page that a user visits. Evaluation results show that the proposed method have low false positive rates and

low overheads. As the future work, we plan to apply the string trace graph for detection of other malicious JavaScript techniques, such as obfuscation. Obfuscation uses a lot of string operations to evaluate string as JavaScript code. String operations in obfuscation will reveal their pattern by string trace graphs.

Acknowledgements. This work was supported by ICT R&D program of MSIP/IITP. [14-824-09-013, Resilient Cyber-Physical Systems Research].

References

1. Alexa top 500 global sites. http://www.alexa.com/topsites
2. Closure compiler. https://developers.google.com/closure/compiler/
3. Exploit database. http://exploit-db.com
4. Malware domain list. http://malwaredomainlist.com
5. The webkit open source project. http://www.webkit.org/
6. Akritidis, P., Markatos, E.P., Polychronakis, M., Anagnostakis, K.: STRIDE: Polymorphic sled detection through instruction sequence analysis. In: Sasaki, R., Qing, S., Okamoto, E., Yoshiura, H. (eds.) Security and Privacy in the Age of Ubiquitous Computing. IFIP, vol. 181, pp. 375–391. Springer, New York (2005)
7. Cova, M., Kruegel, C., Vigna, G.: Detection and analysis of drive-by-download attacks and malicious javascript code. In: Proceedings of the 19th International Conference on World Wide Web, pp. 281–290. ACM (2010)
8. Curtsinger, C., Livshits, B., Zorn, B., Seifert, C.: Zozzle: low-overhead mostly static javascript malware detection. In: Proceedings of the Usenix Security Symposium (2011)
9. Ding, Y., Wei, T., Wang, T., Liang, Z., Zou. W.: Heap taichi: exploiting memory allocation granularity in heap-spraying attacks. In: Proceedings of the 26th Annual Computer Security Applications Conference, pp. 327–336. ACM (2010)
10. Egele, M., Wurzinger, P., Kruegel, C., Kirda, E.: Defending browsers against drive-by downloads: mitigating heap-spraying code injection attacks. In: Flegel, U., Bruschi, D. (eds.) DIMVA 2009. LNCS, vol. 5587, pp. 88–106. Springer, Heidelberg (2009)
11. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The weka data mining software: an update. ACM SIGKDD Explor. Newslett. **11**(1), 10–18 (2009)
12. Polychronakis, M., Anagnostakis, K.G., Markatos, E.P.: Network-level polymorphic shellcode detection using emulation. In: Büschkes, R., Laskov, P. (eds.) DIMVA 2006. LNCS, vol. 4064, pp. 54–63. Springer, Heidelberg (2006)
13. Polychronakis, M., Anagnostakis, K.G., Markatos, E.P.: Emulation-based detection of non-self-contained polymorphic shellcode. In: Kruegel, C., Lippmann, R., Clark, A. (eds.) RAID 2007. LNCS, vol. 4637, pp. 87–106. Springer, Heidelberg (2007)
14. Ratanaworabhan, P., Livshits, V.B., Zorn, B.G.: Nozzle: a defense against heap-spraying code injection attacks. In: USENIX Security Symposium, pp. 169–186 (2009)
15. Rieck, K., Krueger, T., Dewald, A.: Cujo: efficient detection and prevention of drive-by-download attacks. In: Proceedings of the 26th Annual Computer Security Applications Conference, pp. 31–39. ACM (2010)
16. Toth, T., Kruegel, C.: Accurate buffer overflow detection via abstract pay load execution. In: Wespi, A., Vigna, G., Deri, L. (eds.) RAID 2002. LNCS, vol. 2516, pp. 274–291. Springer, Heidelberg (2002)

http://www.springer.com/978-3-319-15086-4

Information Security Applications
15th International Workshop, WISA 2014, Jeju Island, Korea, August 25-27, 2014. Revised Selected Papers
Rhee, K.-H.; Yi, J.H. (Eds.)
2015, XIII, 406 p. 155 illus., Softcover
ISBN: 978-3-319-15086-4