Next Generation Of Exploit Kit Detection By Building Simulated Obfuscators

Tongbo Luo*
Palo Alto Networks Inc
4401 Great America Pkwy
Santa Clara, CA 95054
tluo@paloaltonetworks.com

Xing Jin
Palo Alto Networks Inc
4401 Great America Pkwy
Santa Clara, CA 95054
xijin@paloaltonetworks.com

ABSTRACT

Recently, drive-by downloads attacks have almost reached epidemic levels, and exploit-kit is the propulsion to signify the process of malware delivery. One of the key techniques used by exploit-kit to avoid firewall detection is obfuscating malicious JavaScript program. There exists an engine in each exploit kit, aka obfuscator, which transforms the malicious code to obfuscated code.

In this paper, we tracked and collected over 20000 obfuscated JavaScript samples of 5 exploit kit families from 2014 to 2016. This research is the first attempt to approach the problem from a different angle: reverse engineering the obfuscator from obfuscated samples. We leverage JavaScript normalization technique and hierarchical cluster algorithm to minimize the manual effort when reproducing the obfuscator. We utilize agglomerative approach to measure the proper threshold to best classify samples by their obfuscator version and variant. We implement our design to cluster normalized samples, and illustrate the life cycle of obfuscator version and variant for all 5 families. This is the first work that depicts the timeline of exploit kit from the perspective of the evolution of its obfuscator. We derive patterns on how obfuscator evolved and tend to predict the next obfuscator variation. We share our research with the larger security community through open source the project release, aiming to provide better protection of the cyber-world[1].

Keywords

Exploit Kit, JavaScript Obfuscation, JavaScript Normalization, Obfuscator

1. INTRODUCTION

The cyber space is the number one source of malware (a term that combines "malicious" and "software"), and the majority of these malware threats come from what is called a drive-by download. Attackers developed a toolkit, called exploit kit, that automates the exploitation of client-side vulnerabilities, usually targeting browsers and plugins that a website can access through the browser exposed APIs. Exploit kit packaged all functionalities in 5 stages : entry point, distribution, exploit, infection, and execution, to launch a web attack. The earliest hack toolkit, named Mpack, was available in the crimeware market in 2006. From then on, exploit kits have become the most popular method for cybercriminals to compromise hosts and to leverage those hosts for various methods of profit. Exploit kit can cost anywhere between free to thousands of dollars in underground market. Typically, relatively unsophisticated kit may cost US$500 per month. Licenses for advanced kits have been reported to cost as much as $10,000 per month. Currently, there are 70 different exploit kits in the wild that take advantage of more than a hundred vulnerabilities [1].

A key characteristic of an exploit kit is the heavily obfuscation of their malicious JavaScript code. By virtue of the dynamic feature of JavaScript language (e.g. code within an eval, dynamic type dependent object creation), exploit kit regularly changes the obfuscation techniques to evade detection and/or analysis. Therefore, the world of exploit kits is an ever-changing one, and if people happen to look away even just for one month, they will come back to find that almost everything has changed. This arguably makes detection of exploit kits most pressing problem.

We observed that prior studies focused on extracting features (e.g. static characteristics [2, 3, 4], dynamic behaviors [5, 6] or a combination of both of them [7]) to separate benign and malicious JavaScript code. They usually leveraged machine learning algorithms to train a model based on these features on huge sample set, and adopt the model to classify a sample to either malicious or benign. Till recently, security researchers proposed approaches to detect the variant of an existing exploit kit samples [8, 9, 10]. We have seen lots of articles or blogs [11, 12, 13, 14] that share their analysis on the obfuscation techniques, but they only focus on analyzing the specific techniques utilized by the exploit kits.

In this paper, we approach the problem from a different angle: reverse engineering the obfuscator from the obfuscated script samples. Of the massive number of samples we collected over 2 years, we propose a novel clustering approach to identify the evolution of the obfuscator variant, and significantly reduce the manual effort to reverse engineering the obfuscator to an affordable level.

Our work is motivated by the benefit of reverse-engineering the obfuscator. Purchasing an obfuscator utilized by the real exploit kit is extremely expensive in the underground market, and it may even involve legal issues. This prevents security community from understanding the other side of the obfuscated script. Rebuilding the obfuscator from samples can solve the difficulty of acquiring one in the wild. The rebuilt

*Sr Staff Security Researcher in IPS Team.
[1] Source Code at
https://github.com/irobert-tluo/rebuild_obfuscator.git
Online Detection Tool at http://www.jsDarwin.com

obfuscators can solve the shortage of exploit kit samples as well. Nearly every exploit kit leverages various evasion techniques (e.g. IP cloaking, DGA), which makes consistently sample collecting quite challenging. Using rebuilt obfuscators, we can generate unlimited number of samples.

Reverse engineering an obfuscator is challenging. Unlike develop an arbitrary obfuscator, reverse engineering an obfuscator is more time consuming since we have to manually analyze massive samples to derive how the obfuscator works. However, due to the special functionality of the obfuscator, which is designed to conceal the purpose and logic of the code, the obfuscator deliberately obfuscates the malicious script each time it is requested. Moreover, multiple obfuscator variants are active in the wild at the same time. As a result, from the collected samples, it is extremely difficult to distinguish which obfuscator variant generates a given sample. A feasible approach is required to identify an obfuscator variant and cluster samples in a way that the samples within the same group are generated by the same obfuscator variant. In addition, the samples in each cluster should be as similar as possible to facilitate manual reverse engineering of the obfuscator.

This paper makes the following contributions: (1) *New Angle*: Based on our knowledge, this research is the first attempt to systematically rebuild the obfuscator from the obfuscated samples. (2) *New Techniques*: We developed a new evolution-based hierarchical clustering algorithm to identify obfuscator variants. (3) *Implementation*: We implemented our design and shared our implementation with the security community.

2. BACKGROUND

We begin with an example of obfuscator and obfuscation techniques that gives a high-level overview of how obfuscator works.

2.1 Obfuscation Techniques

Examining the samples we collected in detail, we find that there are three major obfuscation techniques used by obfuscator. (1) Randomization Obfuscation: An obfuscator may randomly insert or modify some elements of the JavaScript program without changing the semantics of the codes, such as whitespace, comments, variable names, function names randomization. They may also change the order of function declarations. (2) Constant Value Obfuscation: An obfuscator utilizes the bracket notation to access a property of an object (e.g. String["fromCharCode"]). However, they construct the property names at runtime as the computation result of other variables or constants. String manipulation is the most common used technique (e.g. s="from" + "Char" + "Code"; String[s];). (3) Encoding Obfuscation: An obfuscator may encode malicious payloads into escaped ASCII characters, unicode, hexadecimal format or even customized representation. They put the encoded malicious payloads and implement the decoding algorithm as the part of the script. The script unpack the encoded payloads and launch the attack.

2.2 Obfuscator

To evade manual analysis, browser emulators, or antivirus detection, adversaries craft their code while remaining effective at exploiting regular users. This keeps the malicious pages for a longer period of time "under the radar" before adding to the public blacklists [8]. Therefore, all exploit kits implement an obfuscator, which leverage obfuscation techniques to deliberately hide the malicious JavaScript program.

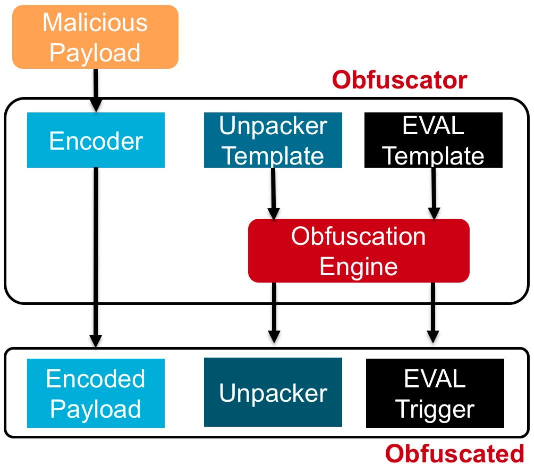
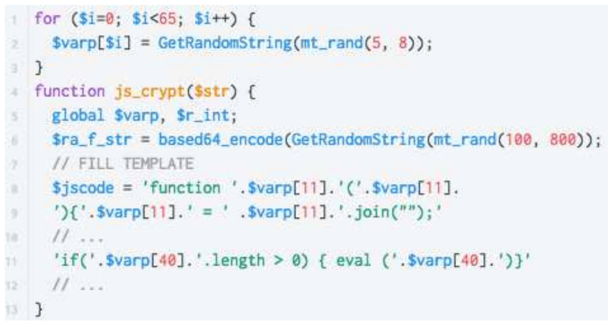


Figure 1: Overview of Obfuscator.

From the released source code of exploit kits, we observe that the design of an obfuscator involves two components: template and engine. Since obfuscators need a convenient way to generate HTML and JavaScript dynamically, they rely on a template to define the static parts of the desired output, as well as, special syntax describing how dynamic content will be integrated. The engine construct the dynamic content (usually a randomized data) and feed it to the template to generate the output. Figure 2 illustrates a snippet of obfuscator code from Nuclear exploit kit [15, 16]. One of the key step to reverse engineering an obfusca-



```
for ($i=0; $i<65; $i++) {
  $varp[$i] = GetRandomString(mt_rand(5, 8));
}
function js_crypt($str) {
  global $varp, $r_int;
  $ra_f_str = based64_encode(GetRandomString(mt_rand(100, 800));
  // FILL TEMPLATE
  $jscode = 'function '.$varp[11].'('.$varp[11].
  '){'.$varp[11].' = ' .$varp[11].'.join("");'
  // ...
  'if('.$varp[40].'.length > 0) { eval ('.$varp[40].')}'
  // ...
}
```

Figure 2: Nuclear Exploit Kit Obfuscator Example.

tor is to identity which part of the sample are generated by the template and which part are generated by the engine. In order to perform the previous step, we need to distinguish whether two samples come from the same variants. In the rest of the paper, we will discuss the method to automatically retrieve the obfuscator template and reduce the number of samples in order to manually reverse-engineering an obfuscator engine.

Figure 1 only shows one level of the obfuscation iterations. An exploit kit normally obfuscates the payload several times. For example, the output of the first level obfuscation will be used as the input of the second level. In this work, we will only demonstrate how to reverse engineering the first level, however our approach can be applied to any level in theory since the obfuscator design at each level is quite similar to each other.

2.3 Challenges

We face several challenges in reverse-engineering the obfuscator:

C1: Code Complexity. Each obfuscated page has more than hundread lines of code, which also contain a lot of random variables inside. This challenge makes it very hard to identity which part of the sample are generated by template and which part are generated by the engine.

C2: Data Complexity. Our sample data set contains more than 20,000 samples over 2 years period. There also may be different versions or variants mixed together at the same time. This challenges make it even harder to do the reverse engineering for the obfuscator.

Despite all the challenges involve, reverse engineering obfuscator would be great help to the researcher/engineer to better understanding and track the evolution of exploit kit

3. OUR APPROACH

In this section, we discuss how we leverage the normalization technique and clustering algorithm to minimize the manual effort when reproducing obfuscators from samples.

3.1 Overview

Figure 3 shows the overview of our approach. First we normalize the JavaScript code to abstract away superficial obfuscation and significantly reduce the number of samples with unique code structure. Then we leverage hierarchical clustering model to cluster the samples so that samples in each cluster are generated by the same obfuscator version or variant. We utilize different merge criterions to incorporate sample to cluster when we identify obfuscator version and variant. These criterions are adopted to simulate the evolution of the obfuscator version or variant during clustering. Therefore, the cluster result reflects the evolution of an obfuscator. We demonstrate how our approach works using the exploit kit samples we have collected in the wild. The data we collected from both public blogs/websites [17] and malicious or compromised servers.

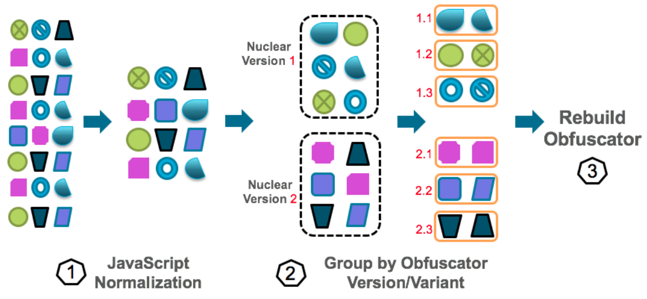


Figure 3: Overview of our approach

We defined some terms we used in this paper to describe obfuscator evolution. We define an obfuscator ***version*** when it changes its template (either EVAL template or Unpack template) and an obfuscator ***variant*** when it changes its engine logic. The major difference between this two types of obfuscator mutation is the impact on the obfuscated code. Obfuscator upgrades to a new version normally involves major changes on the obfuscator side (e.g. new encoding mechanism) and it leads to a different unpacker template which at least takes half of the obfuscated page. On the contrary, an obfuscator variant upgrade causes less impact.

3.2 JavaScript Normalization

We propose to analyze the structure of the JavaScript code rather than the source code itself. The rationale behind normalization is to abstract away randomized information from the obfuscated source code that are irrelevant to the template of obfuscator. As we explained, the obfuscator engine generates random variable names and garbage data that does not change the syntax of the script (e.g. whitespace, comments).

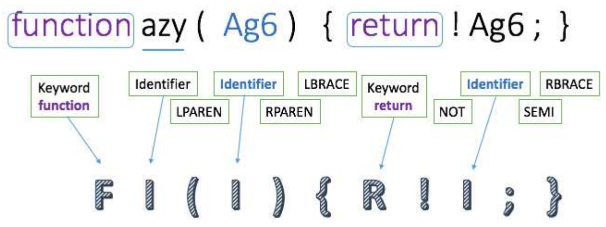


Figure 4: JavaScript Normalization Example.

In order to filter out the randomized data from the obfuscated script, we propose to normalize the script. We utilize a JavaScript lexer to parse the source code and tokenize it into a sequence of tokens. For each type of token, we map it to one unique character. Then we construct the normalized script by concatenating all of the characters together. As Figure 4 shows, keyword *function* and *return* will be converted to character *F* and *R*; symbol *I* represents any variable or identifier regardless of its value; we keep the original content for punctuations (e.g. bracket, comma).

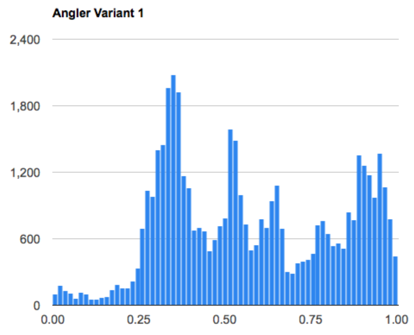
3.3 Normalized Samples

Statistics. In conducting our analysis, we found out that the number of unique sample is significantly reduced after normalization. Table 1 previews our result. For example, among the 7834 Angler examples we have collected, we only identify 613 unique normalized script. In other words, the rest of them share the identical code structure but with randomized content (e.g. variable names).

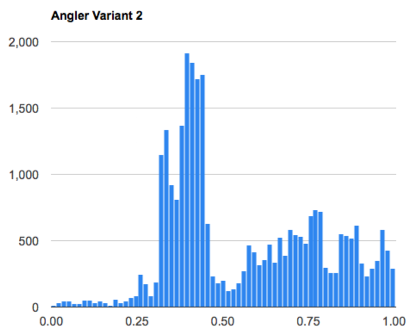
Family	Angler	Nuclear	Rig	KaiXin	Fiesta
TOTAL #	7834	1303	1793	10291	79
UNIQUE #	613	107	344	1543	48

Table 1: Statistics on Normalized Samples

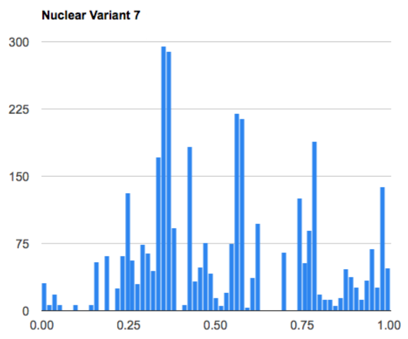
We believe that such massive percentage of identical exploit-kit samples can be leveraged by security researchers. Therefor, we further investigated the duplicated normalized samples. We observed that the majority of them are collected within a short span of time, as described in Figure 6. For example, of the samples from Nuclear family, 91.2 percent of the samples share the same script structure. The chance to find an identical normalized sample off by a day drops to 77.2 percent. For samples collected one week apart, there is only 20 percent chance to find an identical sample.



(a) Angler version 1



(b) Angler version 3



(c) Nuclear version 5

Figure 5: Pairwise Sample Similarity Distribution Angler Varient

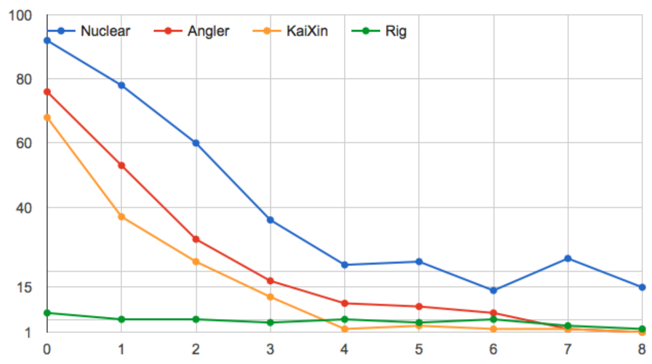


Figure 6: Percentage of Duplicated Normalized Samples within N days sapn

Similarity. Due to the heavy obfuscation used by exploit-kits, the similarity score between the source code of two samples is quite low. We have described how to identify scripts with identical code structure using normalization. We can also compare the similarity of the structure of scripts.

Since normalized script reflects the structure of the script, we can leverage sequence or string similarity algorithm to measure the closeness of scripts in term of their structure. We derive a similarity score from *Levenshtein Edit Distance* which stays in interval [0, 1] using the following formula:

$$SimScore(norm1, norm2) = 1 - \frac{EditDist(norm1, norm2)}{max(len(norm1), len(norm2))}$$

3.4 Cluster Samples by Their Obfuscator

Due to the lack of knowledge on exploit kit, we are not capable of collecting samples along with obfuscator information. Currently, researchers tend to manually analyze a large amount of samples and identify different variants. This process is time-consuming and not reliable since there is not a clearly rule to define new variants. Leveraging the normalization techniques, we significantly reduces the number of unique samples. Since we only filter out the superficial differences in the scripts (e.g. random variable names), the number of unique normalized samples is still quite large for manual analysis. In order to reverse engineering obfuscators, we need to reduce the number of samples to an affordable level. It requires us to group the samples in such a way that samples in the same group are generated from the same obfuscator variant. We then analyze samples within each group and reverse engineering each obfuscator variant.

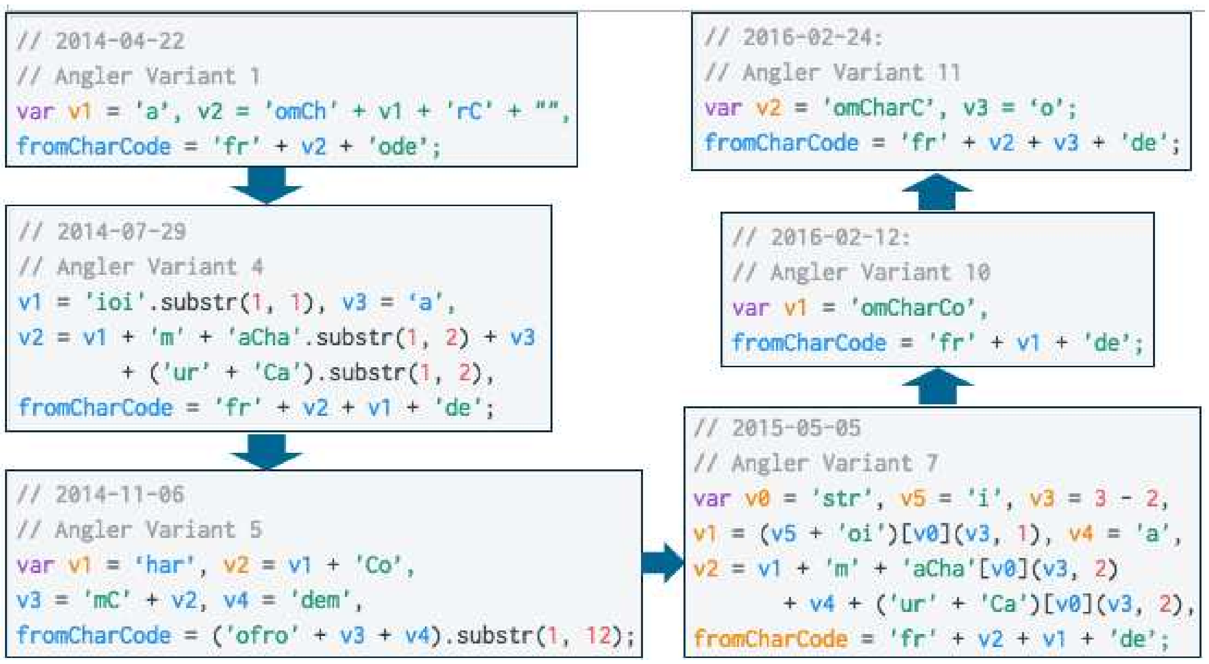
Flat vs Hierarchical. Obviously, clustering algorithm is our best solution to solve the problem. There are two major types of clustering model: flat and hierarchical. Flat model is simple and efficient, but it has several drawbacks. Algorithms in flat model, such as the most popular one K-Means, require a predefined number K as the input which is the number of clusters. This K can be interpreted as the number of obfuscator version in our case. However, it is hard to predict the number of obfuscator version from the samples and it is actually one of the question we wants to answer in our research.

Therefore, we adopt hierarchical clustering model. Unlike the unstructured result returned by flat model, the result from the hierarchical model conveys the structure information. The result can be visualized as a dendrogram, which is a tree diagram to illustrate the arrangement of the clusters. A predefined threshold can be used to cut the dendrogram into subtrees and the leaves in each subtree is a cluster.

Threshold. A dendrogram allows us to adjust the threshold in order to find the best one to identify obfuscator version and variant. We utilize agglomerative (bottom-up) approach to build the dendrogram for each exploit kit family. To measure the threshold, we manually analyzes a couple of samples and marked whether the samples come from the same obfuscator version and obfuscator variants. These samples will be clustered into different places in the dendrogram after we ran the algorithm. We use the marked samples as an anchor to find the proper threshold.

The advantage of hierarchical clustering model model is that we can measure the threshold. The threshold reflects the nature of how an obfuscator evolves, and it is unlikely to be dramatically changed as time goes by (our result spans 2 years). However, the number of new obfuscator versions or variants will definitely increase over time, and we are unable to know it.

Based on our experiment, to identify a new obfuscator **version**, we believe the threshold needs to between **0.4** and **0.5**. Since an obfuscator is designed to conceal the purpose and logic of the code, the obfuscator deliberately make it as different from another at each time the malicious script is generated. The evolution of an obfuscator (e.g. variant with new obfuscation algorithm) happens to lead to the same effect. Although both behaviours could be the explanation of script structure change, we noticed that the latter reason leads to a much significant impact on the changes of



```
// 2014-04-22
// Angler Variant 1
var v1 = 'a', v2 = 'omCh' + v1 + 'rC' + "",
fromCharCode = 'fr' + v2 + 'ode';
```

```
// 2014-07-29
// Angler Variant 4
v1 = 'ioi'.substr(1, 1), v3 = 'a',
v2 = v1 + 'm' + 'aCha'.substr(1, 2) + v3
        + ('ur' + 'Ca').substr(1, 2),
fromCharCode = 'fr' + v2 + v1 + 'de';
```

```
// 2014-11-06
// Angler Variant 5
var v1 = 'har', v2 = v1 + 'Co',
v3 = 'mC' + v2, v4 = 'dem',
fromCharCode = ('ofro' + v3 + v4).substr(1, 12);
```

```
// 2015-05-05
// Angler Variant 7
var v0 = 'str', v5 = 'i', v3 = 3 - 2,
v1 = (v5 + 'oi')[v0](v3, 1), v4 = 'a',
v2 = v1 + 'm' + 'aCha'[v0](v3, 2)
      + v4 + ('ur' + 'Ca')[v0](v3, 2),
fromCharCode = 'fr' + v2 + v1 + 'de';
```

```
// 2016-02-12:
// Angler Variant 10
var v1 = 'omCharCo',
fromCharCode = 'fr' + v1 + 'de';
```

```
// 2016-02-24:
// Angler Variant 11
var v2 = 'omCharC', v3 = 'o';
fromCharCode = 'fr' + v2 + v3 + 'de';
```

Figure 7: Evolution of Angler Obfuscator Version 1

the script structure. Moreover, obfuscator normally changes a large portion of itself (e.g. adopting new data obfuscation algorithms) to make a new variant. This portion (e.g. unpacker/decoder code) is more than 50% of the malicious script. Therefore, our experiment result is reasonable.

To identify a new obfuscator **variant**, the threshold can be set between **0.8** and **0.9**. Each variant only makes minor change on its previous version when evolving, and most of modification is to upgrade its engine logic. For example, the previous variant (say 1.1) engine utilizes concat-based approach to construct sensitive function name *'eval'*(e.g. *a='ev'+'al';*); and the next variant (say 1.2) engine switch to replace-based approach *a='e*al'. replace('*','v');*. Comparing the randomization from the obfuscator engine, the variant changes has a litter more impact on the script structure. This is because polymorphism generated by obfuscator engine normally under certain range (e.g. number of unused garbage variable declaration is in a predefined range). On the other hand, minor obfuscation technique upgrade may comes from developer's random thought that are much irregular.

Within-Cluster Similarity. We measure the pairwise similarity score among samples within each cluster. We find out that the majority of samples within each cluster (for obfuscator version) are not similar to each other. Figure 5 depicts the distribution of the pairwise similarity score for samples generated by Angler version 1, Angler version 3 and Nuclear version 7. The horizontal axis represents the similarity score and the vertical axis represents the number of sample pairs has certain similarity score. We choose them because these obfuscator versions last for a relatively longer time and the number of samples we collected for each cluster is large. The majority of the similarity score is in the range from 0.25 to 0.55. A good clustering model pursues high within-clustering similarity and low inter-cluster similarity. It does not mean our hierarchical clustering algorithm is wrong but the feature of the samples.

The evolution of the obfuscator variant leads to such a diverse script structures among the samples generated by the same obfuscator version. Since for certain obfuscator version, it evolves its obfuscator engine logic to perform polymorphism. The evolution of obfuscator variant are irregular. Due to the dynamic feature of JavaScript, attackers can easily make a new variant with an arbitrary. We monitor the samples generated by the Angler obfuscator version 1 which is active from 2014 to 2016. This obfuscator version keeps evolving, and so far it has 11 variants. We observe that each of the variant utilizes different obfuscation implementations to construct the sensitive API name *fromCharCode*, which is a key method of String object to unpack payload. Figure 7 shows the evolution of obfuscator variants in term of their way to construct string *fromCharCode*.

The *fromCharCode* example and our within-cluster statistics imply a fact that, with the obfuscation variant evolution, the samples changes dramatically. It is possible that samples with similarity score less than 0.2 can be grouped into same cluster. It means security researchers has to keep track of all of the variants in order to name a new version or variant correctly. If they miss the samples for a period of time, even for a week, they may name a new variant which should be an continuation of existing variant. However, our approach can solve this problem.

Life Cycle of Obfuscator Version and Variant. Although the main purpose of our approach is to reproduce obfuscator, it can also be used to identify the life cycle of obfuscators. Since prior reports always leverage the browser/plugin vulnerability targeted by the payload to identify exploit kit changes, we will show the mutation of exploit kit from the evolution of its obfuscator. We use threshold ***0.43*** to identify obfuscator version and use threshold ***0.81*** to identify obfuscator variant. We applied our approach on the sample we collected in real-world and draw a timeline to show the life cycle of each obfuscator.

Figure 8 depicts the timeline of obfuscator version for 5 exploit kit families from 2014 to 2016, and Figure 9 illustrates the timeline of obfuscator variant. Each line in both figures represents an individual obfuscator version, and each block in Figure 9 represents an obfuscator variant.

Different exploit kit family has a diverse life cycle of their obfuscators. Nuclear exploit-kit has constantly evolved from 2009. The life cycle of its obfuscator version has an unique pattern, non-overlapping. It means one obfuscator version died right after a new version get deployed to the server. Obfuscators in the rest of the exploit kit families all has multiple version active in parallel. It is not common that multiple obfuscator variant running at the same time, but we do detect that it is the case for KaiXin exploit kit obfuscator version 1. Angler exploit-kit is the most popular and complicated exploit kit. But it actually has less number of obfuscator version and variant (Figure 9a). Couple of obfuscator versions are active at the same time, especially during the forth quarter of 2015 we collected samples from 4 obfuscator versions. This is may be one of the reasons that people believe it is complex.

Another interesting observation is that, our result matches the fact how exploit kit evolved. According to the report from website WebSense [18], nuclear pack completely replace the older version to a new version at December 2014. And our timeline reflect that the obfuscator also upgraded to a new version.

3.5 Rebuild Obfuscator

Leveraging our clustering algorithm, we cluster samples that generated by a same obfuscator variant into a group. Since the number of unique normalized samples in each group is quite small. By comparing the common structure of the samples in the same group, we can easily rebuild the template used by the obfuscator. For the variable or string whose name or value is randomly mutated, they are server-side variable in the template (e.g. \$varp[11]). Otherwise, it is the fixed content in the template (e.g. '.length > 0)'). The last step is to rebuild the obfuscator logic. It is not hard to reverse engineering the decoding mechanism and implement an encoder. All of these can be easily done since our algorithm cluster the number of samples to an affordable level (average around 15 samples and extremely similar).

4. EVALUATION

We evaluate our cluster result in two ways. Due to the lack of information related evolution of obfuscator, we cannot exactly measure our result. Alternatively, we give the timeline of obfuscator version/variant to the security researchers in our company who focus on exploit kit coverage for several years. Based on their expert knowledge, they believe that is how the obfuscator evolved. We also manually analyzed 50 samples and marked the ones from the same obfuscator version and variant. We checked them in the overall clustering result, and found out that only 2 of them are misplaced.

5. RELATED WORK

Generic Malicious JavaScript Detection ADSandbox [19] proposes to execute suspicious JavaScript in a controlled environment (sandbox). It modified SpiderMonkey, Mozilla's JavaScript engine, to monitor the behavior of JavaScript programs during the runtime. More specifically, ADSandbox intercepts access to every JavaScript object at each time and log it. By applying predefined heuristics (e.g. regular expressions for pattern match) on the resulting logs, the system can detect malicious behaviors.

Zozzle [3] is a static JavaScript malware detector that is fast enough to be used in a browser. Zozzle utilized the browser to extract features from JavaScript AST for both benign and malicious training set. It applied machine learning approach to build the Bayesian classifier model. One of the shortcoming of Zozzle is that it highly depends on the feature robustness of the knowledge model. For the exploit kit scenario,

Environment Detection in Malicious JavaScript Malicious script will try to fingerprint the browser and OS environment, and it will launch the attack only if the environment is vulnerable (e.g. launch attack on IE vulnerability only if the browser is IE with specific version). If the environment is not matched, malware will run the benign code instead. Rozzle [20] focuses on using Symbolic Execution as a way to explore multi-path execution to improve both static and runtime JavaScript detection.

Obfuscation Detection The paper, Caffeine Monkey [21], executes suspicious JavaScript code in a sandbox and recorded the count of each function calls. It marks the script as malicious If the percentage of specific function calls is above a predefined threshold. This paper is the first one to formally address the problem of obfuscation.

Likarsh etc. [2] propose to apply machine learning to the features of obfuscated malicious JavaScript. They extract 65 features in this paper, which include keywords and symbols from static and dynamic analysis. They also apply multiple models to do the classification.

Lu and Debray [22] propose to automatically de-obfuscate JavaScript code, generate the simplified version of it but preserve the semantic as well. They utilize the dynamic slicing algorithm to mark instructions that are relevant to the malware logic.

6. ACKNOWLEDGMENTS

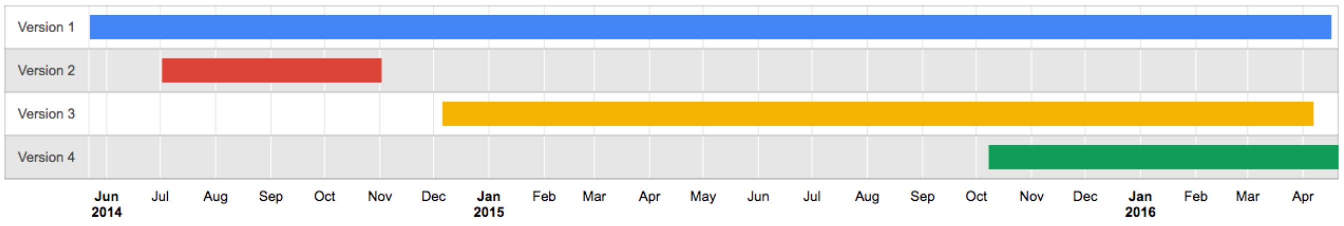
We would like to thank *Xin Ouyang* and *Wei Xu* of Palo Alto Networks Inc for their support on this project and *Rongbo Shao* of Palo Alto Networks Inc for collecting exploit kit samples.

7. REFERENCES

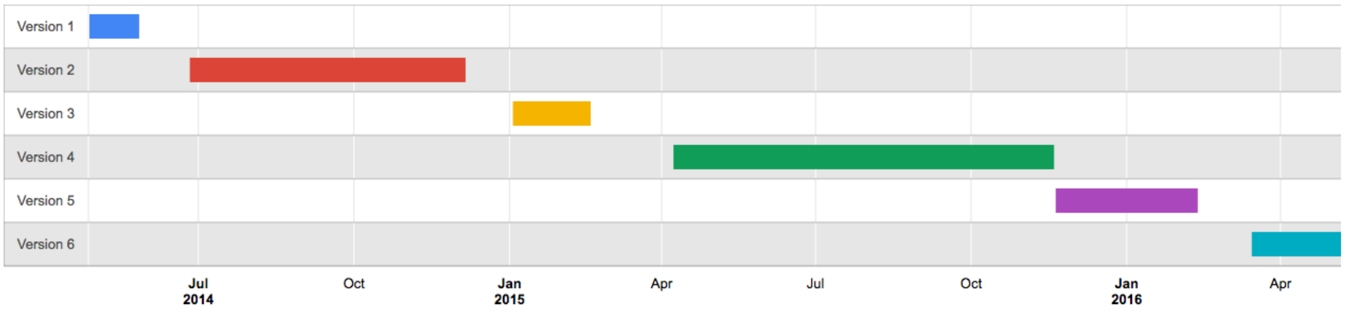
[1] De Maio, G., Kapravelos, A., Shoshitaishvili, Y., Kruegel, C., and Vigna, G., 2014. "Pexy: The other side of exploit kits". In International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Springer, pp. 132–151.

[2] Likarish, P., Jung, E., and Jo, I., 2009. "Obfuscated malicious javascript detection using classification techniques.". In MALWARE, Citeseer, pp. 47–54.

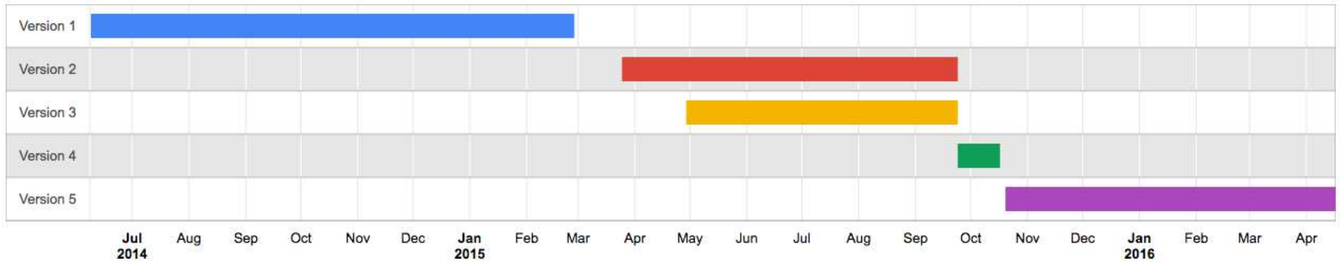
[3] Curtsinger, C., Livshits, B., Zorn, B. G., and Seifert, C., 2011. "Zozzle: Fast and precise in-browser javascript malware detection.". In USENIX Security Symposium, pp. 33–48.



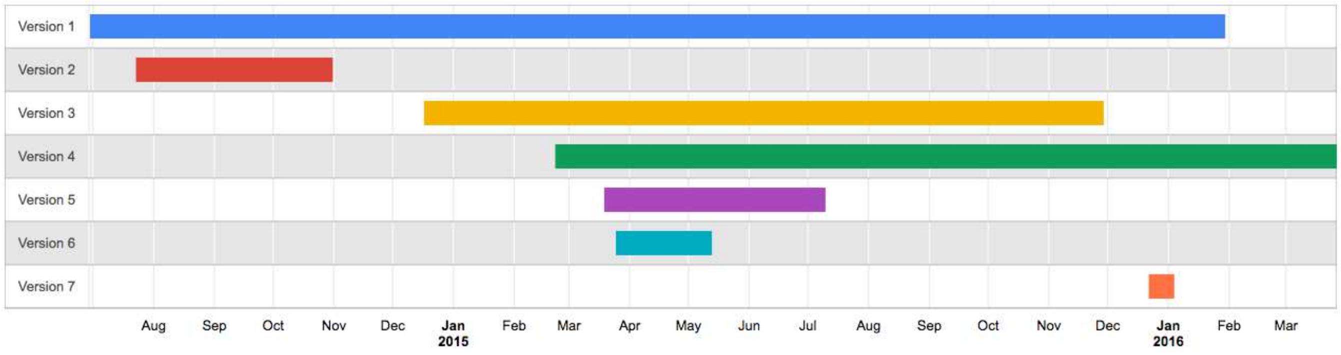
(a) Angler Exploit Kit



(b) Nuclear Exploit Kit



(c) Rig Exploit Kit



(d) KaiXin Exploit Kit

Figure 8: Life-Cycle of Exploit Kit Obfuscator version

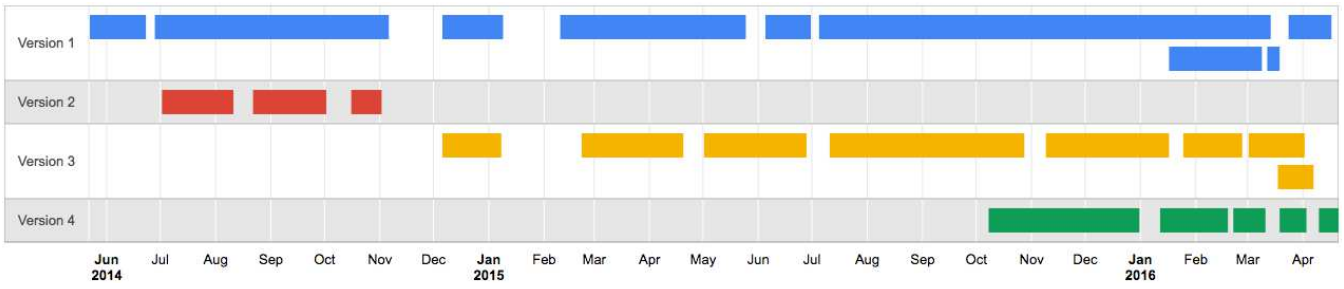
[4] Xu, W., Zhang, F., and Zhu, S., 2013. "Jstill: mostly static detection of obfuscated malicious javascript code". In Proceedings of the third ACM conference on Data and application security and privacy, ACM, pp. 117–128.

[5] Cova, M., Kruegel, C., and Vigna, G., 2010. "Detection and analysis of drive-by-download attacks and malicious javascript code". In Proceedings of the 19th international conference on World wide web, ACM, pp. 281–290.

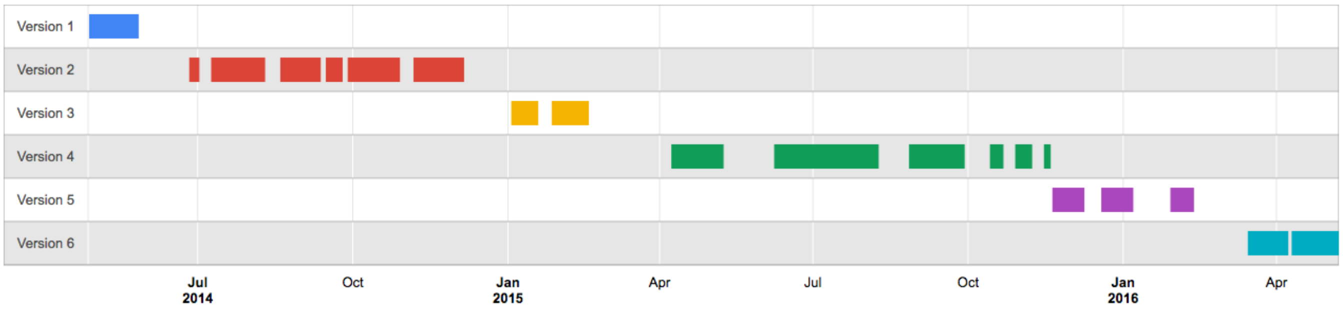
[6] Tzermias, Z., Sykiotakis, G., Polychronakis, M., and Markatos, E. P., 2011. "Combining static and dynamic analysis for the detection of malicious documents". In Proceedings of the Fourth European Workshop on System Security, ACM, p. 4.

[7] Canali, D., Cova, M., Vigna, G., and Kruegel, C., 2011. "Prophiler: a fast filter for the large-scale detection of malicious web pages". In Proceedings of the 20th international conference on World wide web, ACM, pp. 197–206.

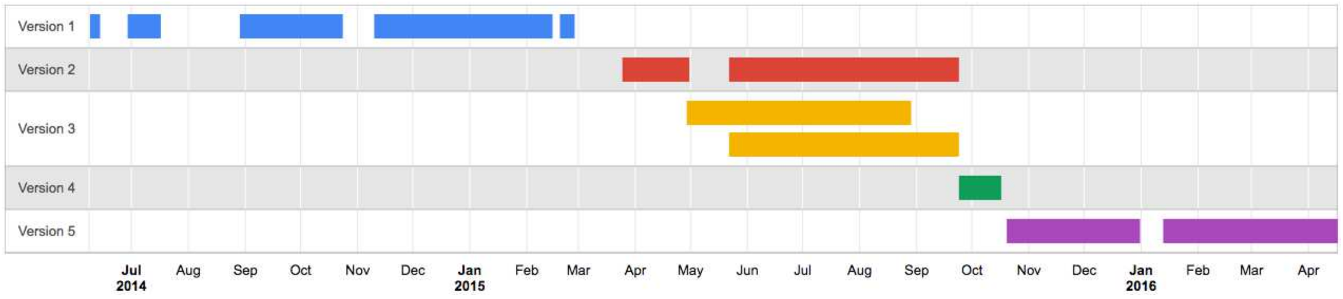
[8] Kapravelos, A., Shoshitaishvili, Y., Cova, M., Kruegel, C., and Vigna, G., 2013. "Revolver: An automated approach to the detection of evasive web-based malware". In Presented as part of the 22nd USENIX



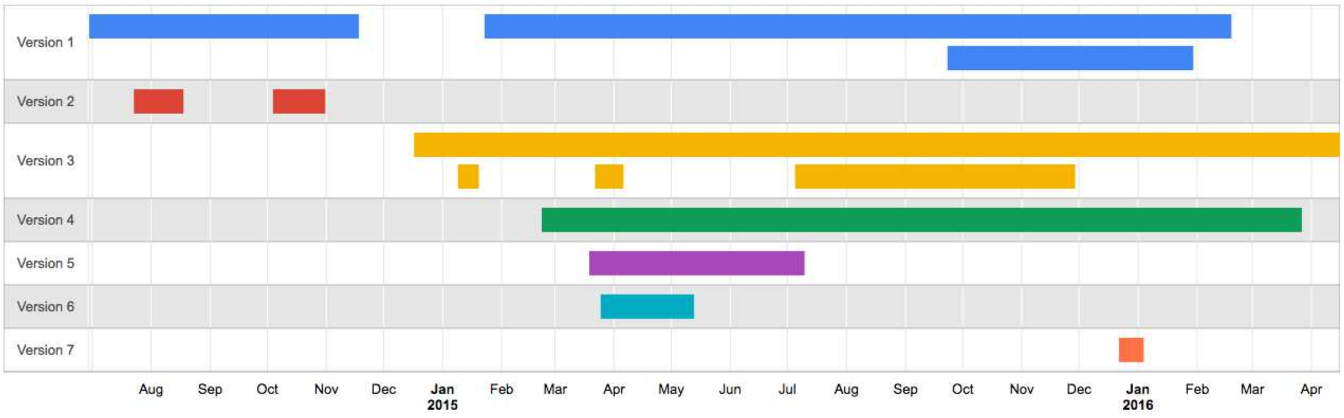
(a) Angler Exploit Kit



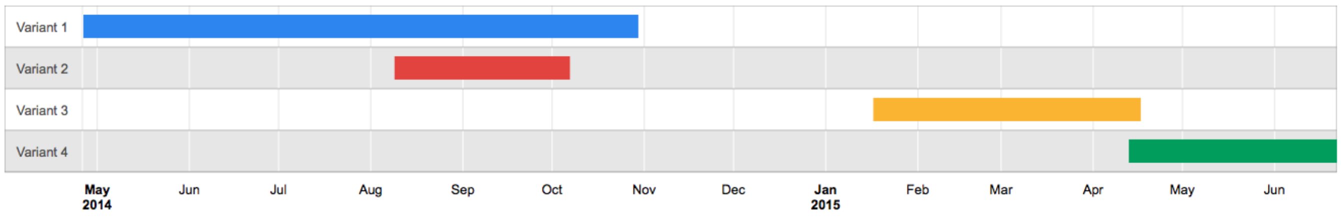
(b) Nuclear Exploit Kit



(c) Rig Exploit Kit



(d) KaiXin Exploit Kit



(e) Fiesta Exploit Kit

Figure 9: Life-cycle of Minor version of Obfuscator variants

Security Symposium (USENIX Security 13), pp. 637–652.

[9] Stock, B., Livshits, B., and Zorn, B., 2015. "Kizzle: A signature compiler for exploit kits". In International Conference on Dependable Systems and Networks.

[10] Taylor, T., Hu, X., Wang, T., Jang, J., Stoecklin, M. P., Monrose, F., and Sailer, R., 2016. "Detecting malicious exploit kits using tree-based similarity searches". In Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy, ACM, pp. 255–266.

[11] Jones, J., 2012. "The state of web exploit kits".

[12] Oliver, J., Cheng, S., Manly, L., Zhu, J., Dela Paz, R., Sioting, S., and Leopando, J., 2012. "Blackhole exploit kit: A spam campaign, not a series of individual spam runs". *Trend Micro Incorporated Research Paper*, pp. 1–19.

[13] Howard, F., 2012. "Exploring the blackhole exploit kit". *Sophos White Paper*.

[14] Xu, W., Zhang, F., and Zhu, S., 2012. "The power of obfuscation techniques in malicious javascript code: A measurement study". In Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on, IEEE, pp. 9–16.

[15] Blog, W. S. L., 2016. Happy nucl(y)ear - evolution of an exploit kit. http://blog.checkpoint.com/wp-content/uploads/2016/04/Inside-Nuclear-1-2.pdf .

[16] Research, S., 2015. Rig exploit kit - diving deeper into the infrastructure. https://www.trustwave.com/Resources/SpiderLabs-Blog/RIG-Exploit-Kit-%E2%80%93-Diving-Deeper-into-the-Infrastructure/ .

[17] Duncan, B. Malware traffic analysis. www.malware-traffic-analysis.net.

[18] Research, C. P. T. I. ., 2016. Inside nuclear's core: Analyzing the nuclear exploit kit infrastructure. http://community.websense.com/blogs/securitylabs/archive/2015/01/15/evolution-of-an-exploit-kit-nuclear-pack.aspx .

[19] Dewald, A., Holz, T., and Freiling, F. C., 2010. "Adsandbox: Sandboxing javascript to fight malicious websites". In Proceedings of the 2010 ACM Symposium on Applied Computing, ACM, pp. 1859–1864.

[20] Kolbitsch, C., Livshits, B., Zorn, B., and Seifert, C., 2012. "Rozzle: De-cloaking internet malware". In 2012 IEEE Symposium on Security and Privacy, IEEE, pp. 443–457.

[21] Feinstein, B., Peck, D., and SecureWorks, I., 2007. "Caffeine monkey: Automated collection, detection and analysis of malicious javascript". *Black Hat USA*.

[22] Lu, G., and Debray, S., 2012. "Automatic simplification of obfuscated javascript code: A semantics-based approach". In Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on, IEEE, pp. 31–40.