Classifying JavaScript malware using Sandbox Execution

Nguyen Quang Vinh
School of Computing
National University of Singapore
nqvinh@u.nus.edu

***Abstract*—JavaScript malware has been prevalent for a very long time, and many methods of detection have been proposed. Most of the methods, however, are ineffective as obfuscation techniques caught up with the technologies. This paper presents Overseer, which employs the technique of using sandbox for JavaScript malware classification. The core of a technique is a browser-like instance where the JavaScript environment is hooked and can capture malware execution information. The approach is hence able to defeat any obfuscation techniques, and using the information generated we can reason about the behavior of malware. Malware classification can be done based on the behavior exhibited.**

***Index Terms*—JavaScript malware, sandbox, malware classification, malware behavior**

I. Introduction

Most browsers nowadays (Chromium-based browsers, Firefox) are always bundled with sandboxed execution of JavaScript, and it is nearly impossible for JavaScript malware to escape the sandbox, then infect the entire system. Instead, JavaScript malware authors take advantage of the Windows Script Host, which allows JavaScript to run natively on Windows. Malware using JavaScript can come in different family categories, with samples recently delivering RATs, ransomwares, cryptojackers. Obviously, to avoid being detected, JavaScript malware authors employ a variety of techniques to obfuscate sensitive data that may be used by antivirus programs. Many JavaScript obfuscation techniques are known by malware analysts. Some malware authors even use JavaScript packers to disguise the payload [1].

A lot of JavaScript malware detection methods have been proposed over the years. One line of work statically analyzes scripts before they are executed, as seen in *Zozzle* [2], *Cujo* [3], *Kizzle* [4], *JStap* [5] and *JaSt* [6]. Another line of work dynamically analyzes scripts, as seen in Revolver [7], by instrumenting the code or via a browser extension. To reduce the runtime overhead resulted from dynamic analysis-based malware detectors, static detectors often serve as a first line of defense. Scripts that are flagged as potentially dangerous by static analysis are put under dynamic analysis for further inspection.

The result is an arms race between increasingly sophisticated obfuscation and evasion techniques on one hand and increasingly effective malware detectors on the other hand. Currently, the most effective malware detection techniques use Machine Learning classifiers to distinguish malicious from benign scripts [2] [3] [4] [5] [6]. These approaches do some feature extraction on a given JavaScript file, e.g n-grams of code tokens, Abstract Syntax Tree features (code contexts, function calls), or simply feed the JavaScript code into a deep neural network to determine whether the file is likely to be malicious. Despite these efforts, cutting-edge static analysis is becoming less effective as obfuscation techniques try to evade the features that the static analysis tools employ. For instance, the *Wobfuscator* tool uses WebAssembly to opportunistically translate carefully selected JavaScript behavior into WebAssembly to evade detection, as all static analysis based method can not parse WebAssembly portion of the code [8]. As a result, the malware file will have its malicious contexts and calls invisible to the malware detectors. Indeed, the tool significantly lowers the detection of known malware samples on static analysis tools listed above [8].

Thus, the dynamic nature of JavaScript makes it very challenging for one to develop a technique to detect, or classify JavaScript malwares. This paper presents the use of a relatively old technique in malware analysis - the use of sandbox execution [9]. Sandboxes offer a restricted system space to run the untrusted suspicious executable files and provide a protected environment to the host system. The malicious files are executed in a jail like environment which doesn't have a access to the host machine network resources, file system and cannot damage the host device. The sandbox is often hooked to log the behavior of the malicious files run in this environment. We can then perform analysis on the logs instead of the usually heavily obfuscated malware sample to have a better view of the behavior of the malware. From such information, we can derive the classification of the malware. This method is resistant to any form of obfuscation, as every sensitive data and function call is clearly revealed.

As this is a progress report, the paper also presents another approach that the author embarks earlier in the NUS Summer Research Program, using Code Property Graphs, a static analysis based idea to reason about JavaScript malware behavior [10].

II. Code Property Graph

A. Overview of Code Property Graph

Code Property Graph (CPG) is a data structure joining Abstract Syntax Trees (ASTs), Control Flow Graphs(CFGs),

and Program Dependence Graphs(PDGs), inheriting the advantages of all code representations. This comprehensive representation seeks to elegantly model templates for common computer software vulnerabilities with graph traversals that can identify things such as buffer overflows, integer overflows, format string vulnerabilities, or memory disclosures [10]. Indeed, the method is very flexible when we try to model different classes of vulnerabilities: syntax-only, control-flow, and even taint-style. It has discovered 88 vulnerabilities in the Linux kernel in 2012 [10]. Later, the authors of CPG expanded the idea and developed *Joern* [11], a Scala-based tool to help bug hunters easily construct queries to find vulnerabilities. The tool covers a variety of programming languages, namely *C/C++*, *x86/x64*, *JVM*, *LLVM Bitcode*, *JavaScript*, *Javasrc*, *Kotlin* and *Python*.

B. Modeling Malware Behavior using CPG

The research initially set out for the idea of modeling malware behavior using CPG queries. Similar to software vulnerabilities, we can leverage the rich information that CPG possess (Control Flow, Function Calls, Syntax Trees, Program Dependence) to statically gain a better insights of the semantics of the program. Indeed, the data structures in CPG are used in compilers. In the world of compilers possible optimizations can be modeled using such structures, some of those are problems like dead code elimination, local value numbering, loop optimizations, and many more [12]. Hence the question, is it possible, to take ideas from problems already defined and researched in software vulnerabilities and compiler design?

The idea is simple enough, by looking at the function calls we can, to some extent, infer the behavior of a malware. A malware which has a lot “encrypt“ calls from a cryptography library and constantly accessing files via “open“ syscalls might infer a ransomware sample. We can map each behavior to the MITRE™ Malware Behavior Catalog [13], and collate all the behaviors to determine the malware family category of any sample. CPG should be flexible enough to model more sophisticated behavior observed in malware, like obfuscation (using sophisticated control-flow related techniques) or sandbox evasion. This idea has some success in languages like Java or Python, and we want to try to expand it to JavaScript.

This is a novel approach, as previous work has only used ASTs and Machine Learning models to evaluate whether a JavaScript file contains malicious code. The downside of this approach, like the authors of *J-Force* [14] pointed out, is that simply looking at information produced from ASTs is not sufficient. We do not have enough information of the code to properly understand its semantics, hence analysis based on such information is ineffective. Indeed, J-Force [14] pointed out a strain of JavaScript malware using collaborative cloaking techniques such as code obfuscation, dynamically created scripts and evasive paths to cleverly hide the semantics behind the dynamic nature of JavaScript. ASTs, even with the assistance of sophisticated Machine Learning models, can not detect malicious code contexts as they do not even appear in the code. Indeed, such samples evade most of the Machine Learning AST-based approach at the time [14]. CPG may solve this problem, as we have richer code information than syntax-related, such as control flow, to model these dynamic obfuscation - which is prevalent in almost all JavaScript exploit kit nowadays. Machine Learning models will also benefit greatly from the additional data provided by CPG as well.

We use the *HynekPetrak* JavaScript malware collection [15] to assist in the modeling of different behaviors of JavaScript malwares. This collection has 39,450 malware samples, and contains a wide variety of malware families [8]. Queries modeling malware behaviors should be general enough to cover all of the behaviors present in all of these samples, but should not be too general to detect a lot of false positives. Also, we need to figure out a way to combine all the malware behaviors to later correctly classify the sample.

C. Difficulties using CPG in JavaScript

As JavaScript is a dynamically typed language, the CPG generated is not closed. This means that much of the execution information may not be clear when we generate the CPG from a JavaScript file. Scripts may run some dynamically determined paths depending on the results of executing different parts in the program. Thus, it is challenging to deal with function call obfuscation techniques (which is very popular in almost all JavaScript malwares) as the only visible portion of code to the CPG is mainly for unrolling the obfuscation. The malicious code can only appear after the deobfuscation has occurred, but the CPG is unable to have any information at this crucial stage for malware identification as no code execution occurs during the generation of CPG. This problem can sometimes make CPG only as effective as AST, and the overhead in runtime is definitely not worth it as we do not really identify crucial information. Other elements in CPG like Program Dependence or Control Flow is often rendered useless as well, due to the aforementioned heavy data obfuscation in JavaScript malware.

Another problem is that tools for converting JavaScript code into CPG representation is not matured yet. *Joern* [11] JavaScript CPG engine maturity is only at ”medium”, and much problems need to be ironed out by the developers later on with parsing different syntax forms in JavaScript. There is a competitor to *Joern* [11], which is initally suggested by the supervisor at the beginning of the research program, the *Fraunhofer-AISEC cpg* [16] tool. Unfortunately, as later we found out, is that the tool can only parse very little amount of JavaScript (more accurately TypeScript) code, and the JavaScript CPG parsing is still in early development. Further research into the use of CPG in modeling vulnerabilities may have to wait for JavaScript code coverage of the tools to become wider, as malwares in the *HynekPetrak* [15] database were starting to use ES2015 syntax in samples dating back to 2017-2018.

The concept of modeling vulnerabilities relies on the fact that we can somehow see the vulnerable code with our human eye, then convert that ”visual” cue into a query. This unfortu-

nately is very tough to do in the realm of JavaScript malware, due to obfuscation one may really struggle to pinpoint the general query to identify a behavior, like opening a port. Cues like function calls or objects are not visible; because of the dynamic nature of JavaScript, malware authors can easily hide everything behind layers of obfuscation. Also, as the JavaScript code is heavily obfuscated, the generated CPG should have thousands of nodes and edges. Simply parsing this much information into a graph data platform will cause a great amount of overhead. The use of *Neo4j* [18] was experimented during the program. Without proper feature extraction or some algorithms to condense the obfuscated code, it is impossible to take a Machine Learning approach to this problem. Training on thousands of samples, each containing thousands of nodes and edges will be very inefficient. Another approach was employing *NetworkX* [19]. There is an observation deduced from analyzing the JavaScript malware, which is that some malware families have very similar structure among their samples(for instance the Nemucod family). This trend is probably due to the popularity of obfuscation tools, hence a lot of samples will share similar obfuscation techniques. However, *NetworkX* cannot parse any CPG produced by *Joern*, as the number of nodes and edges are too large for *NetworkX* to handle. Trimming irrelevant details is much needed for this idea, but due to the lack of time, no solution to effectively trim has been proposed.

Even if we managed to convert the JavaScript code into CPG well, the role of JavaScript in the whole malware chain can be a deterrent in determining the true classification of a malware. Approximately *54%* of the samples in the HynekPetrak malware collection are in the *Downloaders* category [8]. The classifications here should be based on the classification output of different antivirus vendors using signatures (as the malwares in the collection are known and their signatures are already in virus databases of multiple vendors). Antivirus often give classifications based on the final malware payload, as at this stage the attacker's malicious program manifests. Other stages are usually meant for setting up the environment and disabling defense mechanisms for that final payload to activate freely. Hence, CPG can only do as much as classifying what the JavaScript payload do on a behavioral level, and cannot give any information of what the final payload does. Furthermore, further research should define the definition and corresponding set of behaviors of each category of malware as well. There are a lot of definitions of the same malware category in different AV vendors, hence agreeing on a definition should be very useful, and make it easier for others to add, remove or modify a behavior belonging to a category if it is not valid. Tools like *avclass2* can be a good head start.

III. Sandbox execution for JavaScript

A. Overview and Challenges

To deal with the dynamic nature of JavaScript and heavy data obfuscation in JavaScript malware, we can let the malware run in a sandboxed environment, then observe the exhibited behaviors. Sensitive operations, such as opening/writing a file, opening a connection to a server, executing a certain command should be monitored. We can achieve this by simulating the operations that is often used by malware, then hook some logging functionalities. Malware executing in that simulated environment should be the same as executing on a victim's machine, therefore the environment has to simulate as closely to environment in browsers and in the Windows operating system as possible.

This is obviously very tricky. There are a lot of factors to take into consideration when we simulate the browser environment, such as different browsers (with different versions), DOM, common functionalities (like the *window* interface). Simulating Windows has its own set of problems, some are environment variables, Windows Script Host functionalities (Base64, TextStream, FileSystemObject, File System). Also, JavaScript may run in environment that has some libraries inside, ensuring that such dependencies are not missing (one that is most often used is the *prototype.js* library). As mentioned above, JavaScript comes with various forms of syntax that needs to be supported as well. Any of such things missing and the malware will fail to run.

Another problem, but not seen so much in the *HynekPetrak* [15] malware collection, is the existence of sandbox evasion techniques. There exists samples where it tries to get the execution date and time to detect whether it is being ran by a malware analyst or an automated tool. Furthermore, there already exists a lot of techniques in sandbox evasion techniques such as code stalling, fileless malware, fingerprinting behavior. As sandbox execution becomes more popular in automatically analyzing JavaScript malware, sandbox evasion techniques will appear more often in samples. Detecting such attempts and getting rid of evasion code should be taken into consideration.

B. Sandbox Execution log parsing

The structure of Overseer is relatively simple. Malware is executed using a JavaScript sandbox, then the execution logs are parsed into a Python script to detect the behaviors exhibited by the malware in the sandbox. The sensitive operations are logged by the sandbox. Unfortunately, the logs are cluttered with a lot of other details too, such as HTTP response codes from C2 servers, or later stage payloads in ASCII.

There are 16 different *groups* of *signatures*. A signature represents some functionality that is exhibited in the malware logs. A signature belong to the *ActiveX* group, for instance, can represent the activity of "JavaScript creates an ActiveXObject to perform XML HTTP requests". We have *indicators* to make clear what is considered to be under this signature when the Python script parses the logs. A malware behavior may correspond to a single signature, or a combination of different signatures. Modeling the behaviors this way allows for ease of extending or modifying the signature detection, and in turn malware behavior detection as well. Performance is not an issue here, as signature detection can be done in parallel. A thread will take care of a signature group, thus scaling to

include more signatures and groups of signatures should not be a problem.

Signatures should be in a particular order to make them belong to a malware behavior. Another factor is the frequency of the signatures as well - a large number of file accesses should indicate some ransomware-like activity. *Overseer* should take these factors into consideration when evaluating a malware sample.

Malware behaviors identified from the signatures can be used to determine the malware category. For instance, a *Ransomware* may read from file, write to file, and perform cryptography. A *Downloader* on the other hand, may install in system, listen for internet connection, connect to remote server/website, and download payloads. And similar to signatures, malware behaviors in a malware category need to be in a certain order and have appropriate frequencies.

After the malware has been executed in the malware sandbox, artifacts that the sandbox generated (communicated servers, fetched payloads, sandbox logs) are saved into a folder for further inspection and calibration of Overseer later. This way we can get information about other stages' payloads and hashes for AV detections later and malware analysis.

C. Possible extensions and modifications

The tool is inspired from the design of *CybercentreCanada's JsJaws* [20]. *JsJaws* uses *Js-X-Ray* [21] and *NodeJS* sandboxes to evaluate the suspiciousness of a JavaScript file. Signatures are given scores indicating the associated maliciousness. Scores are summed up at the end to decide whether the file has high probability of being a malware.

Machine Learning algorithm is another way to improve the effectiveness of classifying malware. Instead of hard-coding which signature belong to a particular behavior, we can use an AI model. Malware behaviors in a category must be in a order and have appropriate frequency. Hard-coding can become tedious, as there can be large chains of signatures/behaviors, and sometimes the same behavior, like opening a connection to some server, can be done using a variety of Windows functionalities. Sometimes, the JavaScript file just execute a Powershell command to evade detection, and we can employ a AI/ML algorithm to help *Overseer* identify the meaning of such command, which helps figure out the behavior of the malware.

IV. Implementation

Overseer is implemented with Python (v3.9) and the sandbox used is *malware-jail* (v0.19) [22]. Implementation is at https://github.com/IceWizard4902/Overseer. I suggest cloning the original *malware-jail* from Github, then move the build into the *Overseer* folder. The tool is streamlined now, and can correctly determine the signatures in the logs.

Unfortunately, due to the lack of time as this is only started at the end of the project, *Overseer* can only parse the *malware-jail* logs and display the description of the signatures that it detects. Much of the time is spent on correcting the signature indicators.

V. Conclusion

Much work has focused on identifying JavaScript malware using static analysis. We can take some ideas of the compiler design to gain more knowledge into the malware. One such idea is the use of Code Property Graphs, an idea to help model software vulnerabilities. Unfortunately, due to the dynamic nature of JavaScript, the information generated from Code Property Graphs is not easy to make use of, and JavaScript Code Property Graph generators are not matured enough. The paper explores the idea of using sandbox execution to handle heavy data obfuscation in JavaScript. Malware behaviors can easily be identified from the logs of the sandbox, and such information can be leveraged to classify the category of a malware sample.

Acknowledgment

My special thanks to my supervisor, Anis Bin Yusof, for being supportive throughout the NUS Summer Research program. He gives me excellent resources that I should be looking into, and offers meeting to discuss problems I faced every week despite his very busy schedule as a graduate student. Much of the analysis and many of the tools in this paper are from meetings with Anis.

References

[1] L. Tung, "Hackers are disguising their malicious JavaScript code with a hard-to-beat trick," ZdNet, Oct 2021. [Online]. Available at this link

[2] C. Curtsinger, B. Livshits, B. G. Zorn, and C. Seifert, "ZOZZLE: fast and precise in-browser javascript malware detection," in 20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings. USENIX Association, 2011. [Online]. Available: http://static.usenix.org/events/sec11/tech/fullpapers/Curtsinger.pdf.

[3] K. Rieck, T. Krueger, and A. Dewald, "Cujo: efficient detection and prevention of drive-by-download attacks," in Twenty-Sixth Annual Computer Security Applications Conference, ACSAC 2010, Austin, Texas, USA, 6-10 December 2010, C. Gates, M. Franz, and J. P. McDermott, Eds. ACM, 2010, pp. 31–39. [Online]. Available: https://doi.org/10.1145/1920261.1920267

[4] B. Stock, B. Livshits, and B. G. Zorn, "Kizzle: A signature compiler for detecting exploit kits," in 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016, Toulouse, France, June 28 - July 1, 2016. IEEE Computer Society, 2016, pp. 455–466. [Online]. Available: https://doi.org/10.1109/DSN.2016.48

[5] A. Fass, M. Backes, and B. Stock, "JStap: a static pre-filter for malicious JavaScript detection," in Proceedings of the 35th Annual Computer Security Applications Conference, ACSAC 2019, San Juan, PR, USA, December 09-13, 2019, D. Balenson, Ed. ACM, 2019, pp. 257–269. [Online]. Available: https://doi.org/10.1145/3359789.3359813.

[6] A. Fass, R. P. Krawczyk, M. Backes, and B. Stock, "JaSt: Fully Syntactic Detection of Malicious (Obfuscated) JavaScript," in Detection of Intrusions and Malware, and Vulnerability Assessment - 15th International Conference, DIMVA 2018, Saclay, France, June 28- 29, 2018, Proceedings, ser. Lecture Notes in Computer Science, C. Giuffrida, S. Bardin, and G. Blanc, Eds., vol. 10885. Springer, 2018, pp. 303–325.

[7] A. Kapravelos, Y. Shoshitaishvili, M. Cova, C. Kruegel, and G. Vigna, "Revolver: An automated approach to the detection of evasive web-based malware," in Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013, S. T. King, Ed. USENIX Association, 2013, pp. 637–652. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/kapravelos

[8] A. Romano, D. Lehmann, M. Pradel and W. Wang, "Wobfuscator: Obfuscating JavaScript Malware via Opportunistic Translation to WebAssembly," 2022 IEEE Symposium on Security and Privacy (SP), 2022, pp. 1574-1589.

[9] S. Jamalpur, Y. S. Navya, P. Raja, G. Tagore and G. R. K. Rao, “Dynamic Malware Analysis Using Cuckoo Sandbox,“ 2018 Second International Conference on Inventive Communication and Computational Technologies (ICICCT), 2018, pp. 1056-1060.
[10] F. Yamaguchi, N. Golde, D. Arp and K. Rieck, “Modeling and Discovering Vulnerabilities with Code Property Graphs,“ 2014 IEEE Symposium on Security and Privacy, 2014, pp. 590-604.
[11] “Joern“ [Online]. Available: https://github.com/joernio/joern
[12] A. Sampson, “CS 6120: Advanced Compilers: The Self-Guided Online Course“ [Online]. Available: https://www.cs.cornell.edu/courses/cs6120/2020fa/self-guided/
[13] MITRE, “Malware Behavior Catalog“ [Online]. Available: https://github.com/MBCProject
[14] K. Kim, I. L. Kim, C. H. Kim, Y. Kwon, Y. Zheng, X. Zhang, and D. Xu. 2017. “J-Force: Forced Execution on JavaScript.“ In Proceedings of the 26th International Conference on World Wide Web (WWW ’17). International World Wide Web Conferences Steering Committee, pp. 897–906.
[15] HynekPetrak, “javascript-malware-collection“ [Online]. Available: https://github.com/HynekPetrak/javascript-malware-collection
[16] Fraunhofer-AISEC, “cpg“ [Online]. Available: https://github.com/Fraunhofer-AISEC/cpg
[17] malicialab, “avclass“ [Online]. Available: https://github.com/malicialab/avclass
[18] Neo4j, Inc, “Neo4j“ [Online]. Available: https://neo4j.com/
[19] “NetworkX“ [Online]. Available: https://networkx.org/
[20] CybercentreCanada “JsJaws Service“ [Online]. Available: https://github.com/CybercentreCanada/assemblyline-service-jsjaws
[21] NodeSecure “JS-X-Ray“ [Online]. Available: https://github.com/NodeSecure/js-x-ray
[22] HynekPetrak, “malware-jail‘ [Online]. Available: https://github.com/HynekPetrak/malware-jail