A Review on JavaScript Engine Vulnerability Mining

A Review on JavaScript Engine Vulnerability Mining

Zeyan Kang*

School of Information Engineering, Capital Normal University, Beijing, China

*Corresponding author e-mail: kangzeyan@ict.ac.cn

Abstract. With the increasing number of web applications on the Internet, the number of clients increases rapidly. Usually, the client will support the execution of JavaScript language. JavaScript engine has become the core part of modern browser to provide dynamic and interactive website. As of July 2018, about 94.9% of websites use JavaScript language [1]. It makes the browser's JavaScript engine a hot target for attackers. However, due to the characteristics of JavaScript language and inconsistent browser implementation, the vulnerability of JavaScript execution engine has become a major hidden danger of browser security. In this paper, from the composition of JavaScript engine, the common vulnerability forms in the engine, to the existing mainstream engine vulnerability mining tools and methods from dynamic and static perspectives. This paper summarizes the development and existing problems of JavaScript vulnerability mining technology, focuses on the application of fuzzy testing technology in JavaScript vulnerability mining, and analysis the future development trend of JavaScript vulnerability detection combined with existing methods and technologies.

Keywords: JavaScript, Fuzzing, JavaScript Engine, Vulnerability Mining

1. Introduction

The main components of browser include user interface, browsing engine, rendering engine, network, UI back end, data storage, JavaScript engine. The JavaScript engine is used to interpret and execute JavaScript scripts. In the early stage of browser development, JavaScript engine is included in rendering engine. With the development of technology, JavaScript engine gradually becomes independent in browser framework. The four major industrial browsers correspond to different JavaScript engines, including chakra core of IE family, spider monkey of Mozilla Firefox, JavaScript core of Safari and V8 engine of chrome.

JavaScript is essentially a script language for interpreting execution, and its biggest feature is type less. As a result, JavaScript code can not determine the specific variable type information in the compilation stage, and the specific variable type information can only be determined through dynamic execution, which will affect the code execution speed. However, with the advent of JIT (Just-In-Time) technology, the performance of interpretive languages has been improved [2]. The main function of JIT in JavaScript is to convert a part of bytecode with high utilization rate into assembly code when the interpreter works. When hot code is executed again, CPU will skip the interpretation and execute the corresponding assembly code directly to improve the running speed.

Due to the different implementation of JavaScript engine mechanism in different browsers, different types of JavaScript engine are different. When the same code is executed again, different execution results may be produced, which will bring certain security risks. In addition, JavaScript engine has the characteristics of large amount of code, complex structure and complex logic. At the same time, due to the non type characteristics of JavaScript language, it will lead to the error of type derivation confusion when engine designers implement engine logic, especially after introducing JIT mechanism. It may cause attackers to exploit these vulnerabilities for remote attacks. In 2011, IBM released the X-Force report, which tested the security status of JavaScript engine in web applications. The test data included the world's top 500 official websites and other 175 websites. The test results show that 14% of the above websites have JavaScript security vulnerabilities. Through these vulnerabilities, attackers can execute code remotely and implant phishing websites. Moreover, in the follow-up test of IBM, it is found that about 40% of the above websites have JavaScript security problems. JavaScript engine exists in a variety of operating systems, including windows, Linux, IOS, Android, and various IOT devices. In 2018, 55 high-risk vulnerabilities were exposed. The security problem of JavaScript engine has become a big hidden danger that threatens the public network completely. It has been a problem that the academia and industry need to solve in time to excavate the vulnerability of JavaScript engine and timely patch it.

The purpose of this paper is to summarize the existing JavaScript security issues, and to integrate the detection principle and technical implementation of mainstream JavaScript engine vulnerability mining tools in academia and industry. On this basis, an upper framework of vulnerability mining for JavaScript engine is given. In addition to the existing technology, the unsolved security problems of JavaScript engine are analyzed. Combined with the idea of automatic mining, the future technology development trend and direction of JavaScript engine vulnerability mining are given.

2. Classification and Form of Javascript Engine Vulnerability

The whole JavaScript engine is implemented by C / C + + code. Due to the flexible type information of upper level JavaScript language, JavaScript engine developers will cause some logical error loopholes. In addition, there is code self modification in JavaScript engine. The internal logic of the engine can modify the parameters of function, object type and prototype equivalence during code execution. The normal logic should be that after modification, the corresponding logic check should be carried out to ensure that no errors will occur. If you forget to check, it may lead to logic loopholes in the engine.

According to the types of vulnerabilities, JavaScript engine vulnerabilities can be divided into the following categories: reuse after release, buffer overflow, type confusion, tennis game code based on unity, and out of bound interaction between Kinect and unity. On the basis of vulnerability classification, to analyze and understand the formation mechanism of different types of vulnerabilities can deduce how to prevent the corresponding types of vulnerability generation, on this basis, it can provide the direction for JavaScript engine vulnerability mining methods.

2.1. Use-After-Free

The principle of the vulnerability is that the JavaScript engine reuses a pointer (dangling pointer) to this memory after a certain heap memory is released. Dangling pointers refer to a class of pointers that do not point to any legal or valid objects. The result of dangling pointers being referenced is unpredictable because you don't know what will happen. As early as in the 2014 Pwn2Own contest, the reuse vulnerability after the release of the JavaScript engine was used by attackers to bypass the ASLR/DEP protection mechanism of the IE browser, and obtained the ability to execute arbitrary address code on the user side [3]. According to CVE-2019-0568, the InjectJsBuiltInLibraryCode method in the JavaScript engine is used to execute JsBuiltIn.js to initialize some built-in objects [4]. The disable implicit call flag needs to be cleared before calling the JavaScript code, but the implicit call flag is not restored after the call. Since setting the flag can prevent the leakage of objects allocated by the stack, clearing the flag error will cause stack-based use after release. The CrossSite class in

CVE-2018-0946 is used to pass JavaScript variables in different contexts [5]. Firstly, the method covers the virtual function table pointer when wrapping the object, and then rewrites the input and output of the object method, if you directly access the closed context, it may cause reuse after release.

2.2. Stack Overflow

The buffer, also known as the cache, is a part of the computer's memory space. A certain amount of storage space is reserved in the memory space. These storage spaces are used to buffer incoming or outgoing data. This part of the reserved space is called the buffer. The JavaScript engine buffer overflow vulnerability may be caused by operations such as integer overflow and array out-of-bounds access.

For example, in CVE-2018-0758, the method "Lowerer: lowererscontactstrmultiitem" is used to generate the machine code of the connection string [6]. In this function, the InsertAdd operation did not check and caused an integer overflow. In CVE-2016-7202, the implementation of the Array. prototype. reverse function in the ChakraCore engine resulted in an integer overflow after the mathematical operation of the array length, which directly led to a heap-based buffer overflow [7]. In addition, the JavaScript engine is implemented by C++ code. Compared with C++ code, the JavaScript language has some features that can cause buffer overflow errors. For example, in CVE-2017-8636, when constructing a new object, there is a parameter of 0xffff, which will cause a buffer overflow error [8]. This vulnerability can be stably reproduced in the latest Edge of Win10 WIP.

2.3. Type Confusion

The type confusion vulnerability usually takes the data type A as the data type B to resolve the reference, which may lead to illegal access to the data, thus causing the attacker to use the vulnerability to carry out arbitrary address execution attack [9]. For example, in JavaScript, the integer type array is converted into a var type array, and the class object is converted into a data structure. Type confusion vulnerability is common in mainstream browsers, and it is very common in weakly typed programming languages. For example, CVE-2019-0539 vulnerability, which can cause remote code execution when a user visits a malicious web page. The reason is that the dynamically typed object in the JavaScript engine has a property map and a slot array [10]. The attribute mapping is used to obtain the index of the object property in the slot array, and the slot array is used to store the actual data of the attribute. However, CVE-2019-0539 will cause JIT code to confuse objects in memory, which will cause slot array pointer to be covered by arbitrary data.

3. JavaScript Engine Vulnerability Detection Method

Traditional vulnerability detection methods include static analysis methods and dynamic analysis methods. There is no difference between JavaScript engine vulnerability detection and traditional detection methods. Static analysis methods focus on accuracy and the fundamental mechanism of vulnerabilities. The false alarm rate and false alarm rate are low, and the accuracy is high, but it requires a lot of energy to analyze and is too inefficient. Researchers mainly focus on how to rely on existing tools to effectively reduce the time of manual participation. The advantage of the dynamic analysis method is that it pays more attention to the automation in the execution process and does not require too much manual analysis. But relatively, the system has a large overhead, and the false alarm rate and false alarm rate are relatively high. The research focus of the researchers is the accuracy and execution efficiency of the system. Not all JavaScript engines are open source code. For JavaScript engines that can obtain the source code, the address-sanitize method is used when exploiting vulnerabilities to capture memory errors during compilation and throw them after capture. error. Record the corresponding error-causing input samples and errors, and find out the loopholes through subsequent analysis. For JavaScript engines that cannot obtain the source code, tools such as qemu, dynamorio, Frida, etc. are generally used to perform dynamic binary instrumentation operations on the engine to obtain the data status of the engine in real time and receive feedback information.

3.1. Vulnerability Static Analysis Method

Static analysis methods include source code scanning, disassembly scanning and other methods. Source code scanning is mainly for open source programs. By checking the file structure, naming rules, functions, stack pointers, etc. in the program that do not conform to the security rules, the security flaws that may be hidden in the program are discovered. This vulnerability analysis technology requires proficiency in programming languages, pre-defined unsafe code review rules, and check source code through expression matching methods. Since the program is dynamically changing when it is running, if you don't consider the function call parameters and the calling environment, and do not perform lexical analysis and grammatical analysis on the source code, there is no way to accurately grasp the semantics of the program, so this method cannot find the dynamic running process of the program. Security holes in.

Disassembly scanning is often the most effective way to discover security vulnerabilities for programs that do not disclose source code. Analyzing disassembly code requires extensive experience, and auxiliary tools can also be used to help simplify this process, but it is impossible to have a fully automated tool to complete this process. For example, use the disassembler IDA to get the assembly script language of the target program, and then scan the assembled script language to identify some suspicious assembly code sequences. The advantage of finding system vulnerabilities through disassembly is that, in theory, no matter how complicated the problem is, it can always be solved through disassembly. Its shortcomings are also obvious. This method is time-consuming and labor-intensive, and requires high technical level of personnel. It also cannot detect security vulnerabilities generated during the dynamic operation of the program.

Fraser Brown uses static vulnerability detection method to mine vulnerabilities in the process of JavaScript engine compiling intermediate language in browser, and uses micro grammars and language independent analysis strategy to realize vulnerability detection of code binding layer in JavaScript [11]. However, the static analysis method needs to record a lot of instruction information. If the dynamic symbolic execution constraint is needed to be solved, there will be huge overhead and a lot of time consumed, resulting in the reduction of detection efficiency. Sun et al. Proposed shapechecker is a C language symbol execution analyzer using clang as the front end. The spatial and temporal complexity of shapechecker is O(2^n). In order to improve the performance and reduce the overhead, Cha S K proposed to use heuristic search to search the path worthy of exploration, and select a branch in the program, which has many instructions not covered [12]. The complexity of path exploration is reduced by using program analysis methods such as caching function summary and combining with control flow diagram to prune redundant paths.

3.2. Vulnerability Dynamic Analysis Method

Since the execution of the program is a dynamic process, it is not enough to only conduct static analysis during vulnerability mining. The dynamic detection method is to detect the weakness of the program without changing the source code or even the binary code. This type of detection is mainly achieved by modifying the process operating environment. The whole process achieved a high degree of automatic detection of security vulnerabilities. The dynamic method that is often used for JavaScript engine vulnerability mining is fuzzing based on code coverage feedback. Fuzzy testing is an important method in the dynamic vulnerability detection method. It produces a large number of irregular samples that do not conform to the normal code structure or logic. The program executes these deformed samples, collects crash information caused by the deformed samples, and then analyzes the samples and crash information to determine whether there are threatening vulnerabilities. At the same time, when the malformed samples are executed, the program code coverage information is recorded, and the samples that can improve the code coverage are retained as the initial sample template to guide the generation of new samples.

In fuzzy testing, it is commonly used to get the internal operation information of the program by inserting piles. Chen Zhang et al. Proposed a method to mine potential post release reuse type vulnerabilities in JavaScript engine by instrumentation [13]. In this method, the state classification and

transition mode of JavaScript engine execution are defined. AFL (American fuzzy lop), a milestone method in the field of fuzzy testing, is a code coverage bootstrapping tool that supports source code instrumentation [14]. Although it also supports closed source programs based on QEMU, it has poor effect and is prone to errors. Many branch versions are derived from AFL. Aflfast, an improved seed selection algorithm based on AFL, is an accelerated version of AFL, and the fuzzing speed is faster than the original version [15]. Fuzzilli is a JavaScript engine fuzzy testing tool based on syntax variation [16]. Firstly, test cases are generated through syntax template, and then intermediate syntax is generated for mutation. Combined with coverage guidance, more code paths are triggered. Dharma is a fuzzy testing tool based on syntax template generation [17]. It is open-source by Mozilla and used for vulnerability mining of JavaScript engine in Firefox. Chrome fuzzer is a vulnerability mining tool for JavaScript engine of Chrome browser modified based on grinder syntax generator [18]. Statefuzzer and JFF improve the sample generation in fuzzy testing method, and detect the corresponding vulnerabilities according to different types of vulnerabilities [19]. StateFuzzer destroys the original file format and disrupts the execution flow when the sample is generated. It mainly detects the fault tolerance rate of JavaScript engine when processing unconventional file formats. JFF brings the innovation of sample variation by minimizing the variation of existing 1day vulnerability samples. Codealchemist proposes a new test case generation algorithm. The tool can generate arbitrary JavaScript code fragments. These code fragments are correct in semantics and syntax, and it can effectively generate test cases that can cause the JavaScript engine to crash. Table 1 shows a comparison of tools for vulnerability mining of JavaScript engine based on various methods.

Table 1. JavaScript engine vulnerability mining tool

Name	Type	Payload	Efficiency	Accuracy
ShapeChecker	static	fast	high	low
Dachshund	static	fast	high	high
AFLfast	dynamic	fast	high	low
Fuzzilli	dynamic	fast	low	low
Dharma	dynamic	middle	high	high
ChromeFuzzer	dynamic	middle	high	high
StateFuzzer	dynamic	middle	low	high
JFF	dynamic	fast	high	low
CodeAlchemist	dynamic	fast	low	high

4. Conclusion and Future Prospects

At present, the advanced vulnerability detection methods for JavaScript engine mainly focus on generating syntactically correct test cases based on predefined context free syntax or probabilistic language model trained. However, grammatically correct JavaScript statements are often semantically invalid at run time. The development of software protection is always the opposite of attack and defense, attackers are constantly looking for new vulnerabilities to attack programs, and software developers have been improving the protection mechanism of software, and targeted at the means of attackers to protect software. Traditional static and dynamic vulnerability mining strategies have more or less shortcomings. Static analysis method can't get the state of memory and register in real time, and dynamic analysis method can't guide input to mutation in the direction of increasing code coverage. In the future, if we can optimize the mining process by combining dynamic and static methods, it will greatly improve the mining efficiency and mining effect. For vulnerability mining, the quality of samples is very important to the whole mining process. We should consider how to improve the efficiency and accuracy of sample generation, and what kind of generation and mutation strategy can trigger deeper code. Using machine learning and even deep learning, intelligent analysis of existing JavaScript language sample generation rules and logical relations, summarize the rules of sample generation, guide generation and variation, input samples more effectively. In the process of fuzzy testing, we will get a lot of crash information at last. It is time-consuming for us to screen out the real

vulnerabilities in these crash information. We should consider how to reduce unnecessary crashes. Adjust the pile insertion strategy, and obtain the real meaningful feedback information efficiently. Balance the efficiency of plug-in compilation.

References

[1] Information on https://w3techs.com/technologies/details/cp-javascript/all/all
[2] Ha J, Haghighat M, Cong S, et al. A concurrent tracebased just-in-time compiler for JavaScript: TR-09-06[R].Austin: University of Texas, 2009.0
[3] Pwn2Own[EB/OL].[2020- 10- 07]. https://en.wikipedia.org/ wiki/Pwn2Own.
[4] CVE-2019-0568[EB/OL].[2020- 10- 07].https://www.cvedetails.com/cve/CVE-2019-0568/.
[5] CVE-2018-0946[EB/OL].[2020- 10- 07].https://www.cvedetails.com/cve/CVE-2018-0946/.
[6] CVE-2018-0758[EB/OL].[2020- 10- 07].https://www.cvedetails.com/cve/CVE-2018-0758/.
[7] CVE-2016-7202[EB/OL].[2020- 10- 07].https://www.cvedetails.com/cve/CVE-2016-7202/.
[8] CVE-2017-8636[EB/OL].[2020- 10- 07].https://www.cvedetails.com/cve/CVE-2017-8636/.
[9] Meadows C.A procedure for verifying security against type confusion attacks[C]//16th IEEE Computer Security Foundations Workshop, 2003:62-72.
[10] CVE-2019-0539[EB/OL].[2020- 10- 07].https://www.cvedetails.com/cve/CVE-2019-0539/.
[11] Brown F, Narayan S, Wahby R S, et al.Finding and preventing bugs in JavaScript bindings[C]//2017 38th IEEE Symposium on Security and Privacy(SP), 2017:559-578.
[12] Cha S K, Woo M, Brumley D.Program-adaptive mutational fuzzing[C]//2015 IEEE Symposium on Security and Privacy, 2015:725-741.
[13] Hitcon.org[EB/OL].[2020-10-07].https://hitcon.org/2014/downloads/.
[14] Zalewski M.American fuzzy lop fuzzer[EB/OL].(2005). http://lcamtuf.coredump.cx/afl/.
[15] GitHub[EB/OL].[2020-10-07].https://github.com/mboehme/aflfast.
[16] GitHub[EB/OL].[2020-10-07].https://github.com/googleprojectzero/fuzzilli/.
[17] GitHub[EB/OL].[2020-10-07].https://github.com/MozillaSecurity/dharma/.
[18] GitHub[EB/OL].[2020-10-07].https://github.com/demi6od/ChromeFuzzer/.
[19] Fu C Y, Liu W, Ranga A, et al.DSSD:deconvolutional single shot detector[J].arXiv preprint arXiv: 1701.06659, 2017.