Poster: Automatic Dissection of JavaScript Exploits through Dynamic JS-Binary Analysis

Xunchao Hu, Aravind Prakash, Jinghan Wang, Rundong Zhou and Heng Yin
Department of EECS, Syracuse University, Syracuse, New York, USA
Email: {xhu31, arprakas, jwang153, rzhou02, heyin}@syr.edu

I. Introduction

JavaScript exploits impose a severe threat to the computer security. Attacks in browsers, as well as JavaScript embedded within malicious PDFs and Flash documents, are common examples of how attackers launch attacks using JavaScript. A special type of attack called "drive-by-download" makes extensive use of JavaScript and is a major source of infections on the web.

Once a zero-day exploit is captured, it is critical to quickly pinpoint the JavaScript statements that uniquely characterize the exploit and the payload location in the exploit. However, the current diagnosis techniques are inadequate because they approach the problem either from a JavaScript perspective and fail to account for "implicit" data flow invisible at JavaScript level, or from a binary execution perspective and fail to present the JavaScript level view of exploit.

In this poster, we present `JScalpel`, a framework that combines JavaScript and binary level analyses to analyze exploits. It stems from the observation that seemingly complex and irregular JavaScript statements in an exploit often exhibit strong data dependencies in the binary. `JScalpel` utilizes the JavaScript context information from the JavaScript level to perform *context-aware* binary analysis. Further, it leverages binary analysis to account for implicit JavaScript level dependencies arising due to side effects at the binary level. In essence, it performs JavaScript and binary, or *JS-Binary* analysis. Given a JavaScript exploit, our framework performs JS-Binary analysis to: (1) generate a minimized exploit script, which in turn helps to generate a signature for the exploit, and (2) precisely locate the payload within the exploit. It replaces the malicious payload with a friendly payload and generates a PoV for the exploit.

II. Overview

A. Problem Statement

We aim to develop `JScalpel`– a framework to combine JavaScript and binary analyses to aid in analysis of memory corruption exploits.

Input: `JScalpel` accepts a raw exploit and a vulnerable program as input. The exploit consists of HTML and malicious JavaScript components. The exploit can be obfuscated or encrypted. `JScalpel` makes no assumptions about the nature of payloads. That is, the payload could be ROP-only, executable-only or combined.

Output: `JScalpel` performs JS-Binary tracing and slicing and generates 3 specific outputs. (1) A simplified exploit HTML that contains the key JavaScript statements that are required to accomplish the exploit, and (2) the precise JavaScript statements that inject the payload into the vulnerable process' memory along with the exact payload string within the JavaScript. Finally, (3) an HTML page, where the malicious payload is replaced by a benign payload is generated as a Proof-of-Vulnerability (PoV).

B. `JScalpel`– *Overview*

Figure 1 presents the architecture of `JScalpel`, which leverages Virtual Machine Monitor (VMM) based analysis. It consists of the following 4 key components.

a) Multi-level Tracing: In order to perform effective analysis of script based attacks, it is essential to not only observe the program execution at a script level, but also comprehend the corresponding effects at a binary level. The vulnerable program is exploited within the VM. Meanwhile, a JavaScript trace is gathered within the victim process' context using JavaScript debugger interface. The JavaScript tracer selectively turns the binary tracer on such that the binary trace precisely captures the instructions that are executed within the context of the currently executing JavaScript statement.

b) Exploit and Payload Detection: `JScalpel` uses a CFI component to detect exploits. Specifically, the execution is monitored and when an exploit is detected, the corrupted pointer is noted as a slicing-source for exploit simplification. This source is the entry point for exploitation. Next, the exploit is allowed to continue and successive violations are noted. Each violation occurs due to execution of a ROP-gadget and serves as a separate slicing-source for non-executable payload. If at some point, memory allocated by the process is executed, the entry point is noted as the slicing-source for executable payload entry point.

c) Multi-level Slicing: Multi-level slicing comprises of binary- and JavaScript-level slicing. First, starting from each of the slicing sources identified by the CFI module, the trace is sliced to obtain key binary instructions. From the context information obtained from the JavaScript tracer, the JavaScript statements corresponding to the binary instructions are retrieved. These JavaScript statements form the slicing-sources for JavaScript slicing. Next, backward slicing is performed at the JavaScript-level to generate the simplified script. To ensure that the simplified script is functional, both control- and

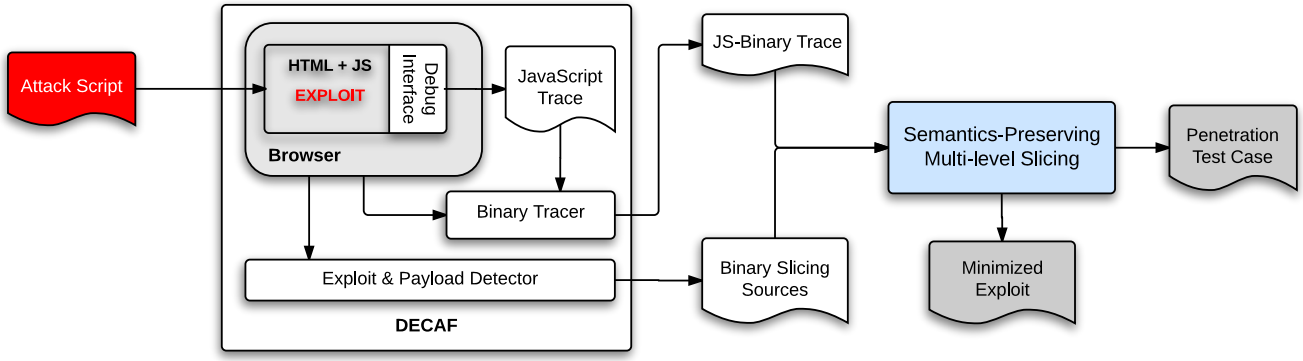

Fig. 1: Architecture of JScalpel

TABLE I: Exploit Analysis Results

CVE	# Unique JS stmts	# JS slicing sources from binary analysis	# Stmts from JS analysis only	Can Stmts from JS-only analysis cause crash?	# Stmts from JS-Binary analysis	Are exploit-semantics preserved?	Size reduction
2009-0075	30	9	6	✓	17	✓	43.3%
2010-0249	45	3	6	✗	19	✓	57.7%
2010-0806	803	2	10	✓	10	✓	98.8%
2010-3962	105	1	1	✓	1	✓	99%
2011-1255	97	40	1	✗	16	✓	83.5%
2012-1876	67	32	1	✗	30	✓	55.2%
2012-1889	77	1	2	✓	2	✓	97.4%
2012-4969	117	16	1	✗	8	✓	93.2%
2013-3163	43	9	1	✗	13	✓	69.8%
2013-3897	187	26	1	✗	41	✓	78.1%

data-dependencies are taken into account while performing the JavaScript-level slicing. Furthermore, the slicing can also identify the string payload and segregate the payload into non-executable and executable payloads.

d) Generating Minimized Exploit and Proof-of-Vulnerability (PoV): The JavaScript level slice readily yields the simplified script for a given source. Therefore, simplified exploit and the payload injecting statements are the slices corresponding to slice source from first CFI violation and payload-related slice-sources respectively. The HTML code is extracted from the original exploit HTML page and repackaged along with the simplified exploit.

Payloads are often encoded within strings in an exploit. The runtime value during decryption can be compared against the payload content identified by the CFI module. This yields the exact offsets of non-executable and executable payloads within the string. The executable payload is replaced by a benign payload to generate the PoV.

III. Implementation & Evaluation

We implemented JScalpel prototype on top of DECAF [2]. The code comprises of 890 lines of Python, 2300 lines of Java and 4000 lines of C++ code. Table I presents the evaluation of JScalpel on a corpus of 10 exploits, 9 from Metasploit [1] and 1 wild exploit. With pure JavaScript level analysis, only 4 of 10 exploits can be minimized and generate functional exploits . However, with our new JS-Binary analysis technique, we were able to minimize the exploit-specific statements by about 77.6% on average, and precisely identify the payload, in a semantics-preserving manner, meaning that the minimized exploits are still functional. In each of the exploits tested, we replaced the payload and generated a Metasploit test case.

IV. Conclusion

We presented JScalpel, a framework that combines JavaScript and binary analyses to analyze JavaScript exploits. Our multi-level tracing bridges the semantic gap between JavaScript level and binary level to perform dynamic JS-Binary analysis. By performing multi-level slicing from the sources identified by CFI violation, JScalpel is able to determine the payload injection and exploitation statements of the JavaScript exploits. We analyze 9 recent memory corruption exploits from Metasploit and 1 exploit from the wild and successfully recover the payload and a minimized exploit for each of the exploits.

References

[1] Metasploit: Penetration testing software. http://www.metasploit.com/.

[2] Henderson, A., Prakash, A., Yan, L. K., Hu, X., Wang, X., Zhou, R., and Yin, H. Make it work, make it right, make it fast: Building a platform-neutral whole-system dynamic binary analysis platform. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2014), ISSTA 2014, ACM, pp. 248–258.