Vulnerability Study in WebAssembly Runtimes

Sara Baradaran, Jungkyu Kim, Ritu Pravakar, Yutian Yan

February 2024

Abstract

WebAssembly runtimes provide a sandboxed environment that abstracts away the underlying hardware and provides a consistent execution environment for WebAssembly code. The presence of vulnerabilities in a runtime not only changes the code behavior when executing but also can get exploited by attackers to gain control of systems relying on the runtime. Understanding the root causes of vulnerabilities in runtimes allows early detection of them. In this project, we intend to collect and analyze previously detected vulnerabilities in WebAssembly runtimes both quantitatively and qualitatively. Moreover, we will categorize these vulnerabilities based on their root causes. We also aim to find potential vulnerability patterns and offer suggestions to the WebAssembly community for fixing those patterns before leading to vulnerability introduction.

1 Introduction

WebAssembly has emerged as a binary format and a portable target for compiling code written in high-level programming languages such as C/C++, Rust, and Go [1]. WebAssembly runtimes serve as sandboxed execution environments for WebAssembly applications. There exist two types of WebAssembly runtimes, web browser engines and standalone runtimes.

Web browser engines, such as Google Chrome's V8, Mozilla Firefox's SpiderMonkey, and Microsoft Edge's Chakra, incorporate WebAssembly support in their overall functionality. Specifically, they provide an integrated environment for running WebAssembly modules within web browsers. On the other hand, standalone runtimes are independent implementations of WebAssembly engines designed to be used outside web browsers, catering to diverse application scenarios beyond the web, such as command-line tools, desktop applications, and embedded systems.

In recent years, many vulnerabilities have been detected in WebAssembly runtimes, which enable attackers to take control of execution in these runtimes, thereby seriously harming relying systems and users. However, there is a notable absence of systematic studies on vulnerabilities specific to WebAssembly runtimes within our community. By comprehensively understanding detected vulnerabilities, developers can gain insights into common vulnerable code patterns, functions, and APIs, enabling them to avoid similar pitfalls when designing and implementing analogous software. Moreover, since many vulnerabilities often share the same root causes, delving into the root causes of known vulnerabilities can unveil undiscovered security vulnerabilities.

In this project, we propose to conduct a thorough examination of past vulnerabilities in WebAssembly runtimes to identify recurring patterns and root causes, with the ultimate goal of improving the security posture of WebAssembly-based systems. Through systematic analysis of vulnerabilities, we aim to provide actionable recommendations and guidelines to developers, enabling them to build more robust and secure WebAssembly applications and runtimes.

2 Related Work

Analyzing Bugs in WebAssembly Runtimes. Some researchers have recently studied bugs in WebAssembly runtimes [2, 3]. In the following we breifly review these studies.

Wang et al. [2] conducted an empirical analysis of more than 800 reported bugs across four WebAssembly runtimes. They analyzed bugs regarding their root causes, symptoms, bug-fixing time, and the number of files and lines of code involved in the bug fixes. Through this study, Wang et al. found that Incorrect Algorithm Implementation is the most prevalent root cause, and Crash is the most common symptom of bugs present in WebAssembly runtimes. Moreover, at the median, the bug-fixing time is less than 14 days. They also observed that more than half of bug fixes involve only one file, and on average, less than 40 lines of code need to be changed for bug fixes.

Likewise, Zhang et al. [3] studied 311 bugs in three WeAssembly standalone runtimes. They categorized these bugs into 31 categories and summarized their common fix strategies. They observed that studied bugs often have specific patterns and share similarities. Hence, Zhang et al. developed a pattern-based bug detection framework to identify bugs in WeAssembly runtimes. This framework specifically generates test cases specific to each bug category. For example, bug-triggering file operations include renaming, moving, counting, and mapping. Hence, the framework tests if a WebAssembly runtime can correctly rename a file or report error information when the file does not exist. Otherwise, it shows the presence of a bug related to file-handling operations.

Bugs primarily affect functionality and usability, while vulnerabilities directly impact the security of software systems. Though bugs are worth considering, addressing vulnerabilities is typically of higher priority due to their potential for confidentiality, integrity, and availability breaches. In this research, we target analyzing WebAssembly runtime vulnerabilities.

Analyzing Vulnerabilities in Virtual Machines. Some other researchers have targeted vulnerability study in virtual machines (VMs) [4].

In [4], Yilmaz et al. assert that inconsistent, erroneous, or ambiguous vulnerability information hinders the identification and resolution of vulnerabilities in web-based virtual machines. For example, a large portion of ActionScript vulnerabilities are ambiguously classified as *memory corruption* vulnerabilities by the CVE database. However, a deeper investigation shows they can be classified as stack overflow, heap overflow, use-after-free, double-free, and integer overflow vulnerabilities. In the following, the authors reclassified ActionScript CVE vulnerabilities labeled as generic *memory corruption* and *unspecified* into one of more fine-grained sub-classes. They also proposed Inscription, a method for transforming Adobe Flash binary code, which enables protection against different categories of Flash vulnerabilities without modifying vulnerable Flash VMs.

3 Preliminary Study

As a data collection process, we have collected a set of 113 known Common Vulnerabilities and Exposures (CVEs) in WebAssembly runtimes from 2016 to 2024. These vulnerabilities are getting exploited due to the presence of one or multiple security weaknesses in the codebase of runtimes. Each security

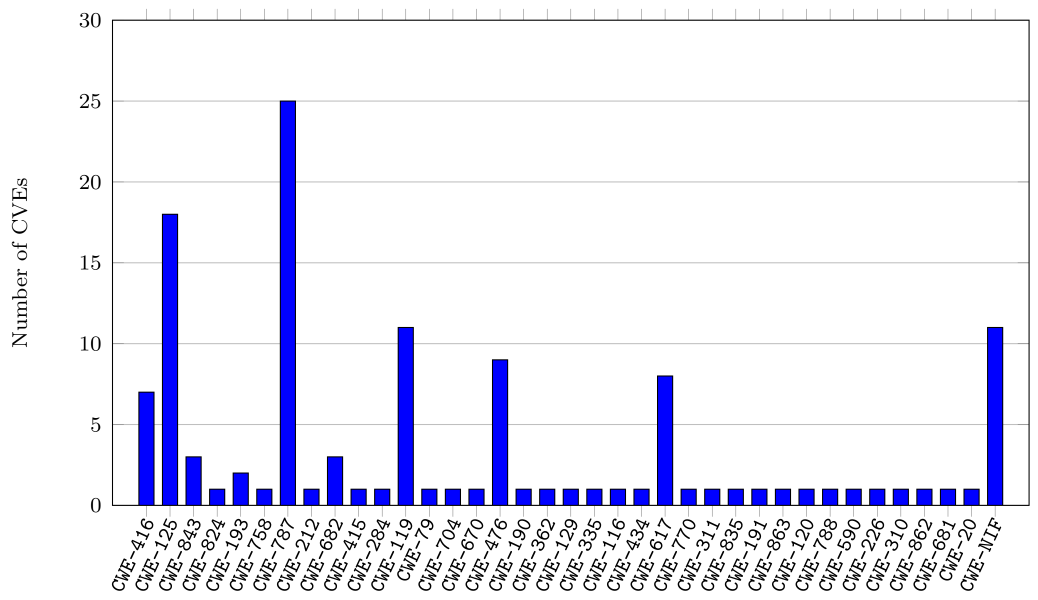

Figure 1: Frequency distribution of CWEs

weakness is often shown by an ID in the Common Weakness Enumeration (CWE), a taxonomy for identifying the common sources of software flaws. Fig. 1 demonstrates the distribution of different weaknesses in CVEs. Note that the last column labeled `CWE-NIF` refers to those CVEs for which there is no adequate information regarding the type of weaknesses. According to Fig. 1, 36 known weaknesses are responsible for introducing vulnerabilities in WebAssembly runtimes. Moreover, only six types of weaknesses, including `CWE-787`: out-of-bounds memory write, `CWE-125`: out-of-bounds memory read, and `CWE-119`: improper restriction of operations within the bounds of a memory buffer, `CWE-476`: NULL pointer dereference, `CWE-617`: reachable assertion, and `CWE-416`: use after free are responsible for 68% of the whole vulnerabilities for which there is sufficient information regarding the type of weaknesses.

3.1 Sample CVE Analysis

As illustrated, the most common root weakness leading to vulnerability is out-of-bounds memory write. Thus, we have selected `CVE-2022-39394` [5] as a CVE example to demonstrate how a vulnerability in a runtime might get

```c
typedef uint8_t wasmtime_trap_code_t;

WASM_API_EXTERN bool wasmtime_trap_code(const wasm_trap_t *,
                                        wasmtime_trap_code_t *code);
```

Figure 2: `wasmtime_trap_code` declaration in `trap.h`

```rust
1 /* vulnerable function in versions 2.0.1 and earlier */
2 pub extern "C" fn wasmtime_trap_code(raw: &wasm_trap_t, code: &mut i32) ->
       bool { /* function body */ }
3
4 /* patched function in versions 2.0.2 and later */
5 pub extern "C" fn wasmtime_trap_code(raw: &wasm_trap_t, code: &mut u8) ->
      bool { /* function body */ }
```

Figure 3: `wasmtime_trap_code` definition in `crates/c-api/src/trap.rs`

exploited by an out-of-bounds memory write. This CVE affected Wasmtime version 2.0.1 and earlier. Wasmtime [6] is a standalone WebAssembly runtime that prioritizes speed and security and is usable both as a command-line utility and as a library embedded in an application.

In versions before 2.0.2, there is a vulnerability in Wasmtime's C API implementation where the definition of the function `wasmtime_trap_code` does not match its declared signature in the header file `trap.h`. As declared in Fig. 2, this function, as the second parameter, receives the address of a variable whose type is defined as `uint8_t`, which typically represents a 1-byte unsigned integer. However, in line 2 of Fig. 3, `wasmtime_trap_code` takes a mutable reference to an `i32` variable as the second parameter. The `i32` type represents a 4-byte signed integer. By passing a mutable reference, the function can modify the value stored at the memory location pointed to by `code`. Thus, calling `wasmtime_trap_code` results in a 4-byte write into a 1-byte buffer provided by the caller, leading to three zero bytes being written beyond the 1-byte location. In line 5 of Fig. 3, the patched function is shown that properly takes a mutable reference to an `u8` variable, which represents a 1-byte unsigned integer.

Attackers may take advantage of an out-of-bounds memory write and execute malicious code by modifying the program's execution flow so that the subsequent instruction points to a memory location where the malicious code is injected. When the next instruction is executed, the malicious code will be executed instead of the expected code, leading to a successful control hijacking attack. An out-of-bounds memory write also paves the way for denial-of-service (DoS) attacks. It can be exploited to overwrite key data

```
1  static void get_error_message(const char *message,
2                                wasmtime_error_t *error, wasm_trap_t *trap)
3  {
4      fprintf(stderr, "error: %s\n", message);
5      int sensitive_data = 7189273;
6      wasmtime_trap_code_t code;
7      wasm_byte_vec_t error_message;
8      if (error == NULL)
9      {
10         /* wasmtime_trap_code() will write 4 bytes into the 1 byte
       wasmtime_trap_code_t */
11         wasmtime_trap_code(trap, &code);
12         fprintf(stderr, "trap code: %u\n", code);
13         wasm_trap_message(trap, &error_message);
14         wasm_trap_delete(trap);
15     }
16     else
17     {
18         wasmtime_error_message(error, &error_message);
19         wasmtime_error_delete(error);
20     }
21     fprintf(stderr,"%.*s\n", (int)error_message.size, error_message.data);
22     wasm_byte_vec_delete(&error_message);
23     /* some other code ... */
24 }
```

Figure 4: A function calling vulnerable API `wasmtime_trap_code`

structures, causing the program to malfunction or crash.

A short code snippet is shown in Fig. 4 to illustrate how using vulnerable `wasmtime_trap_code` API function can jeopardize the security aspects of a program employing it. According to this figure, one can write a function `get_error_message` that is called when an error concerning a WebAssembly module occurs in Wasmtime (e.g. when instantiating a module fails). This function can provide a description of an error and the trap code associated with a given trap. However, calling `wasmtime_trap_code` in line 11 causes 4-byte to be written in a 1-byte variable `code`, thereby leading to the value of adjacent variable `sensitive_data` to be overwritten. This data overwriting can subsequently change the flow of the program or cause a program crash.

4 Future Work

In the following, we intend to quantitatively and qualitatively analyze CVEs in more detail, as shown in Section 3.1 by an example. Then, we will categorize vulnerabilities based on their root causes. We also aim to find vulnerability patterns in WebAssembly runtimes and provide constructive suggestions to the WebAssembly community to assist them in developing more robust execution environments.

References

[1] "Webassembly." [Online]. Available: https://webassembly.org

[2] Y. Wang, Z. Zhou, Z. Ren, D. Liu, and H. Jiang, "A comprehensive study of webassembly runtime bugs," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2023, pp. 355–366.

[3] Y. Zhang, S. Cao, H. Wang, Z. Chen, X. Luo, D. Mu, Y. Ma, G. Huang, and X. Liu, "Characterizing and detecting webassembly runtime bugs," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 2, dec 2023.

[4] F. Yilmaz, M. Sridhar, A. Mohanty, V. Tendulkar, and K. W. Hamlen, "A fine-grained classification and security analysis of web-based virtual machine vulnerabilities," *Computers & Security*, vol. 105, p. 102246, 2021.

[5] "CVE-2022-39394: out-of-bounds write vulnerability in a Wasmtime's C API function named wasmtime_trap_code." [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2022-39394

[6] "Wasmtime: A fast and secure runtime for WebAssembly." [Online]. Available: https://wasmtime.dev