An Empirical Study of Vulnerabilities in WebAssembly Runtimes

Sara Baradaran

Abstract—**WebAssembly runtimes provide a sandboxed environment that abstracts away the underlying hardware and consistently executes WebAssembly code. The presence of vulnerabilities in a runtime not only changes the code behavior at execution time 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 and helps developers avoid similar vulnerabilities when developing analogous software. In this paper, we have collected and analyzed a set of XX known vulnerabilities in WebAssembly runtimes both quantitatively and qualitatively. Through this study, we observed that many vulnerabilities share the same root causes. We classified the root causes into XX different categories and investigated the distribution of each root cause. We also found that XX reported vulnerabilities are still present and have not been patched by developers, which affects the security of systems that use vulnerable runtimes.**

Index Terms—**WebAssembly, Runtime, Vulnerability Study, CVE Analysis**

I. Introduction

Since 2017, 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 allows developers to execute computationally intensive tasks, such as image and video processing, cryptography, and data manipulation in near-native performance. While WebAssembly was initially designed for running code in web browsers, its applications have expanded beyond the web domain in blockchain [2], cloud computing [3], and Internet of Things (IoT) [4]. As WebAssembly is increasingly adopted for various applications, there is a growing ecosystem of WebAssembly runtimes. WebAssembly runtimes serve as sandboxed environments for executing WebAssembly applications. These runtimes can be broadly classified into two main categories: web browser engines and standalone runtimes. Browser-based engines, such as Google Chrome's V8 [5], Mozilla Firefox's SpiderMonkey [6], and Microsoft Edge's Chakra [7], incorporate WebAssembly support in their overall functionality. Specifically, they provide an integrated environment for running WebAssembly modules within web browsers. Table I demonstrates the most widely-used WebAssembly engines used in browsers.

Runtime	Source	Browser
V8	C++	Google Chrome
SpiderMonkey	C++/Rust/JavaScript	Mozilla Firefox
Chakara	C/C++/JavaScript	Microsoft Edge
JavaScriptCore	C/C++	Apple's Safari

TABLE I
BROWSER-BASED RUNTIMES FOR WEBASSEMBLY

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. There exist various projects on open-source platforms (e.g., GitHub), as shown in Table II, which are developed as standalone execution environments for WebAssembly [8], [9].

Typically, there exist three execution modes for running WebAssembly code, including just-in-time (JIT) compilation, ahead-of-time (AOT) compilation, and interpretation. Each standalone WebAssembly runtime may support one or more execution modes. Just-in-time (JIT) compilation is an execution technique that dynamically translates the bytecode or intermediate representation of a program into native machine code during runtime, just before it is getting executed. The on-the-fly compilation process optimizes the code for the specific hardware platform on which it is running by leveraging runtime information and profiling data. In contrast, ahead-of-time (AOT) compilation involves precompiling WebAssembly code into machine code before execution. The compiled native code is typically stored as a disk file and loaded into memory when the WebAssembly code needs to be executed. By eliminating the need for dynamic compilation during runtime, AOT compilation helps mitigate the performance overhead associated with JIT compilation. It allows for faster startup times and reduces memory consumption compared to JIT compilation.

Interpreters are another group of WebAssembly runtimes. A WebAssembly interpreter is a software component specifically designed to directly execute WebAssembly bytecode instead of first translating it into machine code. It operates by reading the binary format of a WebAssembly module and sequentially executing the instructions it contains. An interpreter may rely on a set of functions written in high-level languages, which are invoked to emulate the desired functionality when a WebAssembly instruction needs to be executed.

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 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,

Runtime	Source	LoC	Stars	Active	Execution
GraalWasm	Java	66.5k	19.7k	Yes	?
Wasmer	Rust	225.9k	17.6k	Yes	AoT/JIT
Wasmtime	Rust	370.4k	14.3k	Yes	AoT/JIT
WasmEdge	C/C++	98.9k	7.8k	Yes	AoT/JIT
Wasm3	C/C++	15.0k	7.0k	Yes	Int
Lunatic	Rust	13.7k	4.5k	Yes	JIT
Wazero	Go	151.4k	4.5k	Yes	AoT/Int/?
WAMR	C/C++	172.9k	4.4k	Yes	AoT/JIT/Int
Lucet	Rust	30.2k	4.1k	No	AoT
Extism	Rust	6.4k	3.6k	Yes	?
WAVM	C/C++/Python	66.1k	2.6k	No	JIT
Life	Go	5.6k	1.7k	No	Int
Wasmi	Rust	68.3k	1.3k	Yes	Int
Wagon	Go	16.2k	902	No	Int
VMIR	C/C++	50.2k	651	No	JIT/Int
Wac	C/C++	3.6k	464	No	Int
aWsm	C/C++/Rust	386.3k	282	Yes	AoT
WasmVM	C/C++	11.9k	210	Yes	Int
Fizzy	C/C++	27.3k	206	No	Int
WasmKit	Swift	21.8k	$\leq$200	Yes	Int
py-wasm	Python	12.4k	$\leq$100	No	Int
Swam	Scala	20.6k	$\leq$100	No	Int
Warpy	RPython	3.2k	$\leq$100	No	JIT

TABLE II
STANDALONE RUNTIMES FOR WEBASSEMBLY

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. Understanding the security issues of runtimes can also notify users about security concerns of the underlying runtimes they rely on.

In this research, we conduct a thorough assessment of known 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. This study answers the following research questions.

RQ1: What are the root causes of vulnerabilities in WebAssembly runtimes and how they are distributed? Root causes help researchers gain insight into the nature of vulnerabilities. To answer this question, we first categorized root causes for vulnerabilities based on a systematic process, and XX root causes are identified. We then analyze the root cause distribution of these vulnerabilities.

RQ2: Do different vulnerabilities detected in a runtime share the same root causes? Identifying the common root causes of different vulnerabilities can help security testers focus on software components more likely to contain vulnerabilities. The existence of vulnerabilities with similar root causes can also suggest that there might be systemic flaws in the design, implementation, or configuration of the runtime.

RQ3: How many of legacy vulnerabilities has been already patched? We tried to reproduce XX vulnerabilities using proof of concept (PoC) files provided by people who reported vulnerabilities for the first time. We found noticeable variations in the degree of responsibility within the developers community when it comes to fixing vulnerabilities detected in different runtimes. [TODO: We can add more RQs in the next updates]

II. Data Collection

A. Selection of WebAssembly Runtimes

Besides four widely-used browser-based engines that support WebAssembly code execution, we inspect WebAssembly runtimes projects on GitHub using the curated awesome-wasm lists [8], [9] that include more than 30 WebAssembly runtimes currently available. As shown in Table II, these standalone runtimes are written in different programming languages, which provide different levels of language safety. A group of runtimes are actively maintained by developers, while the remaining are no longer maintained (the rows colored in gray). If the most recent commit in a project's repository occurred over a year ago, we assume the project has reached the end of its lifecycle. In some cases, the project's developers have explicitly mentioned that the project is no longer under maintenance. For example, developers of Lucet switched focus to working on the Wasmtime engine in mid-2020. They asserted that all of Lucet's features are already ported to Wasmtime.

In this study, we focus on sufficiently mature runtimes, whether they are now active or not. Specifically, we pruned out small projects with less than 3k LoC, which are developed as experimental runtimes and only support a minimal specification of WebAssembly. As a result, such projects mentioned in awesome-wasm lists have not been included in Table II.

B. Collection of Vulnerabilities in WebAssembly Runtimes

To investigate the characteristics of WebAssembly runtime vulnerabilities, we first crawled XX Common Vulnerabilities and Exposures (CVEs) from the National Vulnerability Database (NVD) using the keywords "WebAssembly" and "Wasm". Then, we filtered out XX CVEs out of XX, as these vulnerabilities are associated with non-runtime WebAssembly tools (e.g., Binaryen, WABT) designed for compiling, optimizing, and debugging WebAssembly code.

Besides using the NVD dataset, we looked for issues and pull requests whose content included keywords such as "vulnerability" and "security". We used the GitHub REST API [10] to build the crawler and retrieved XX reported vulnerabilities across XX different WebAssembly runtimes.

III. Study of Vulnerabilities in WebAssembly Runtimes

A. RQ1: Vulnerability Root Causes

We investigate vulnerability information to identify and analyze the types of root causes among the issues. Specifically, for each reported vulnerability available in the NVD database, we read the vulnerability description alongside the conversation on the GitHub issue's page (if available) to find out the underlying reason resulting in the vulnerability. Through this process, we found the following XX root causes for WebAssembly runtime vulnerabilities.

Missing of WebAssembly input validation. WebAssembly input validation refers to the process of verifying the correctness and safety of WebAssembly modules before execution. It involves checking the structure, types, and instructions within the Wasm module to ensure they adhere to the WebAssembly specification [11]. In this study, we observed that a notable number of vulnerabilities have been introduced in WebAssembly runtimes since a group of runtimes do not validate input modules before executing them. For example, Wasm3 currently assumes that input Wasm files are already validated. Though this runtime performs many checks before executing the input, the Wasm3 validation process is incomplete yet. As a result, Wasm3 gets almost many fuzzing-detected vulnerabilities, the majority of which are produced by invalid WebAssembly inputs. In our collected dataset, 11 vulnerabilities have been reported in Wasm3, which all share the same root cause, the incomplete validation of WebAssembly input modules.

Implementation bugs for WebAssembly new features. Since the advent of WebAssembly, several proposals have been proposed to enhance and expand WebAssembly's capabilities. We found that many vulnerabilities are introduced in WebAssembly runtimes due to the bugs in implementing the newly proposed features. In the following, we review some of these features and the vulnerabilities they cause.

a) Reference Types: This proposal adds support for reference types in WebAssembly, which aims to enable more efficient communication between the host and the Wasm module. Without the reference types, WebAssembly cannot take or return references to the host's objects (e.g., DOM node on a web page). Instead, one can store the host objects in a side table and refer to them by index. However, this adds an extra indirection, and implementing the side table requires cooperating glue code on the host side. The glue code is also troublesome since it is host-specific and outside the Wasm sandbox, diluting Wasm's safety guarantees. In this way, if one intends to use the Wasm module on the web, in a Rust host, and in a Python host, three separate glue code implementations are needed, causing WebAssembly to become less attractive as a universal binary format. Thus, the value type *externref* removes the need for glue code to interact with host references. The proposal for reference types has three main parts. First, a new *externref* type that represents an opaque, unforgeable reference to a host object. Second, an extension to WebAssembly tables, allowing them to hold *externrefs* in addition to function references. Third, new instructions for manipulating tables and their entries. With these new capabilities, Wasm modules can directly talk about host references rather than requiring external glue code running in the host. The *externref* should not violate the WebAssembly's sandboxing properties. Therefore, it is supposed to provide two features opaqueness and unforgeability. Opaqueness means that a Wasm module cannot observe an *externref* value's bit pattern. Since *externref* is completely opaque from the module perspective, the only way to use it is to send an *externref* back to the host as an argument of an imported function. Passing a reference to a host object into a Wasm module also does not reveal any information about the host's address space and the layout of host objects within it. The *externref* is also unforgable, meaning that a Wasm module cannot create a fake host reference out of thin air. An *externref* cannot be dereferenced by the module, and the module cannot directly access or modify the data behind the reference. Indeed, the module cannot even be sure which kind of data is being referenced. It can only return either a reference we already gave it or a null reference.

CVE-2021-39216 [12] refers to a vulnerability in Wasmtime that is triggered when the host passes multiple *externrefs* to a WebAssembly instance at the same time. If Wasmtime's VMExternRefActivationsTable[1] became full after passing in the first *externref*, then a garbage collection will occur when the second *externref* is passed. However, the first *externref* is not protected from being collected and could become free. Later, when control is given to the WebAssembly content, it could still try to use the first *externref* even though it has been freed. In addition to this CVE, we found four other CVEs associated with the Wasmtime, which stem from implementation bugs in the management of *externrefs*.

b) SIMD (Single Instruction, Multiple Data): SIMD is a proposal that aims to introduce native support for vector operations in WebAssembly, enabling the parallel process of data. SIMD instructions are a specific class of instructions that exploit data parallelism in applications by simultaneously performing the same operation on multiple data elements. Computationally intensive applications such as audio/video code and image processors take SIMD instructions to accelerate performance. The high-level goal of the WebAssembly SIMD proposal is to introduce vector operations to the WebAssembly specification in a way that guarantees portable performance. The WebAssembly SIMD proposal defines a portable, performant subset of SIMD operations available across most modern architectures. The current proposal is limited to standardizing fixed-width 128-bit SIMD operations. It introduces a new v128 value type, used to represent different types of packed data, and a number of new operations that operate on this type (on packed data). To determine these operations, some criteria are considered. First, the operations should be well-supported across multiple modern architectures. Second, performance wins should be positive across multiple relevant architectures within an instruction group. Finally, the chosen set of operations should minimize performance cliffs if any.

CVE-2023-41880 [13] has been assigned to a vulnerability in the Wasmtime, which is triggered when using the `i64x2.shr_s` instruction (stands for "shift right signed") with a constant shift amount larger than 32 bits on x86-64 platforms. The `i64x2.shr_s` WebAssembly instruction performs a right shift operation on each lane of a 128-bit vector of two signed 64-bit integer values (i64). This instruction takes two operands: a 128-bit vector of two i64 values (referred to as

[1]A data structure used to manage the lifetime of external references passed to WebAssembly modules.

a lane) and a 64-bit scalar value (referred to as a shift amount). It shifts each lane of the vector to the right by the number of bits specified by the shift amount, considering the sign of the values. This bug causes the instruction to produce an incorrect result, bringing about an off-by-one error (OBOE) vulnerability[2] The vulnerability happens due to a miscompilation of `i64x2.shr_s` instruction by the Wasmtime's Cranelift code generator. Overall, we found three vulnerabilities in Wasmtime, coming from incorrect machine code generation of instructions in WebAssembly SIMD proposal. [TODO: We will find and add more root causes for the next updates]

IV. Related Work

Analyzing Bugs in WebAssembly Runtimes. Some researchers have recently studied bugs in WebAssembly runtimes [14], [15]. Wang et al. [14] 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, they found that Incorrect Algorithm Implementation is the most prevalent root cause, and program 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. [15] studied 311 bugs in three WeAssembly standalone runtimes. They categorized these bugs into 31 categories and summarized their common fix strategies. They observed that these 311 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 studies in virtual machines (VMs) [16]. In [16], 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. Specifically, the authors reclassified ActionScript vulnerabilities labeled as generic *memory corruption* and *unspecified* into one of the 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.

[TODO: Since we are still analyzing the GitHub issues for collecting vulnerabilities, we will report the exact number of vulnerabilities in our dataset in future reports. In this version we only use XX instead of exact numbers]

[2]The low 32-bits of the second lane of the vector are derived from the low 32-bits of the second lane of the input vector instead of the high 32-bits.

References

[1] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," *SIGPLAN Not.*, vol. 52, no. 6, p. 185–200, jun 2017. [Online]. Available: https://doi.org/10.1145/3140587.3062363

[2] W. Chen, Z. Sun, H. Wang, X. Luo, H. Cai, and L. Wu, "Wasai: uncovering vulnerabilities in wasm smart contracts," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 703–715. [Online]. Available: https://doi.org/10.1145/3533767.3534218

[3] M. N. Hoque and K. A. Harras, "Webassembly for edge computing: Potential and challenges," *IEEE Communications Standards Magazine*, vol. 6, no. 4, pp. 68–73, 2022.

[4] E. Wen and G. Weber, "Wasmachine: Bring iot up to speed with a webassembly os," in *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, 2020, pp. 1–4.

[5] "V8: Google chrome's javascript and webassembly engine." [Online]. Available: https://v8.dev

[6] "Spidermonkey: Mozilla's javascript and webassembly engine." [Online]. Available: https://spidermonkey.dev

[7] "Chakracore: Microsoft edge's javascript and webassembly engine." [Online]. Available: https://github.com/chakra-core/ChakraCore

[8] "Awesome webassembly runtimes." [Online]. Available: https://github.com/jcbhmr/awesome-webassembly-runtimes?tab=readme-ov-file

[9] "Awesome webassembly runtimes." [Online]. Available: https://github.com/appcypher/awesome-wasm-runtimes

[10] "GitHub REST API." [Online]. Available: https://docs.github.com/en/rest/search/search?apiVersion=2022-11-28

[11] "WebAssembly Specification." [Online]. Available: https://webassembly.github.io/spec/core

[12] "CVE-2021-39216." [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2021-39216

[13] "CVE-2023-41880." [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2023-41880

[14] 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.

[15] 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.

[16] 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.