WebAssembly

Subjects: Computer Science, Software Engineering
Contributor: Håkon Harnes , Donn Morrison

WebAssembly is a low-level bytecode language that enables high-level languages like C, C++, and Rust to be executed in the browser at near-native performance. WebAssembly has gained widespread adoption and is natively supported by all modern browsers. Despite its benefits, WebAssembly has introduced significant security challenges, primarily due to vulnerabilities inherited from memory-unsafe source languages. Moreover, the use of WebAssembly extends beyond traditional web applications to smart contracts on blockchain platforms, where vulnerabilities have led to significant financial losses. WebAssembly has also been used for malicious purposes, like cryptojacking, where website visitors' hardware resources are used for crypto mining without their consent.

WebAssembly cryptojacking smart contracts security vulnerabilities

1. Introduction

The Internet has come a long way since its inception and one of the key technologies that have enabled its growth and evolution is JavaScript. JavaScript, which was developed in the mid-1990s, is a programming language that is widely used to create interactive and dynamic websites. It was initially designed to enable basic interactivity on web pages, such as form validation and image slideshows. However, it has evolved into a versatile language that is used to build complex web applications. Today, JavaScript is one of the most popular programming languages in the world, currently being used by 98% of all websites [1].

Despite its popularity and versatility, JavaScript has some inherent limitations that have become apparent as web applications have grown more complex and resource-demanding. Specifically, JavaScript is a high-level, interpreted, dynamically typed language, which fundamentally limits its performance. Consequently, it is not suited for developing resource-demanding web applications. To address the shortcomings of JavaScript, several technologies, like ActiveX [2], NaCl [3], and asm.js [4], have been developed. However, these technologies have faced compatibility issues, security vulnerabilities, and performance limitations.

WebAssembly was developed by a consortium of companies, including Mozilla, Microsoft, Apple, and Google, as a solution to the limitations of existing technologies. WebAssembly is designed as a safe, fast, and portable compilation target for high-level languages like C, C++, and Rust, allowing them to be executed with near-native performance in the browser. It has gained widespread adoption and is currently supported by 96% of all browser instances [5]. Moreover, WebAssembly is also being extended to desktop applications [6], mobile devices [7], cloud computing [8], blockchain virtual machines (VMs) [9][10][11], IoT [12][13], and embedded devices [14].

Overview. WebAssembly is a technology that aims to address performance, compatibility, and security issues that have plagued previous approaches. It was developed by a consortium of tech companies, including Mozilla, Microsoft, Apple, and Google, and was released in 2017 [15]. WebAssembly has since gained widespread adoption and is currently supported by 96% of all browser instances [5]. Additionally, it is an official World Wide Web Consortium (W3C) standard [16], and is natively supported on the web. An overview of WebAssembly is given in **Figure 1**.

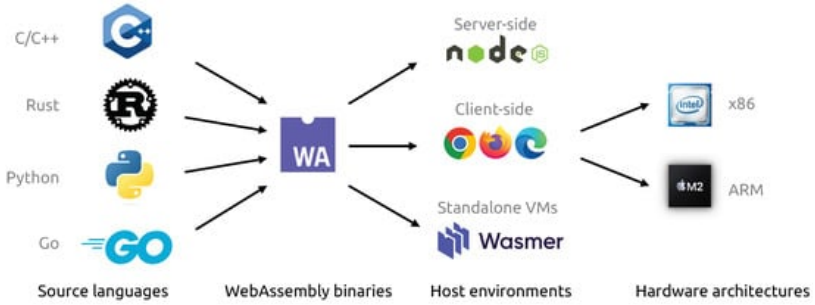

Figure 1. WebAssembly serves as the intermediate bytecode bridging the gap between multiple source languages and host environments. The host environments compile the WebAssembly binaries into native code for the specific hardware architecture.

WebAssembly is a low-level bytecode language that runs on a stack-based Virtual Machine (VM). More specifically, instructions push and pop operands to the evaluation stack. This architecture does not use registers; instead, values are stored in global variables that are accessible throughout the entire module or in local variables that are scoped to the current function. The VM manages the evaluation stack, global variables, and local variables.

Host Environment. WebAssembly modules run within a host environment, which provides the necessary functionality for the module to perform actions such as I/O or network access. In a browser, the host environment is provided by the JavaScript engine, such as V8 or SpiderMonkey. WebAssembly exports can be wrapped in JavaScript functions using the WebAssembly JavaScript API [17], allowing them to be called from JavaScript code. Similarly, WebAssembly code can import and call JavaScript functions. Other host environments for WebAssembly include server-side environments like Node.js [18] and stand-alone VMs with accompanying APIs. For instance, the WebAssembly System Interface (WASI) [19] allows WebAssembly modules to access the file system.

Module. WebAssembly modules serve as the fundamental building blocks for deployment, loading, and compilation. A module contains definitions for types, functions, tables, memories, and globals. In addition, a module can declare imports and exports, as well as provide initialization through data and element segments or a start function.

Compilation. Languages like C, C++, and Rust can be compiled into WebAssembly since it is designed as a compilation target. Toolchains like Emscripten [20] or wasm-pack [21] can be used to compile these languages to WebAssembly. The resulting binary is in the `wasm` binary format, but can also be represented in the equivalent human-readable text format called `wat`. A module corresponds to one file. The WebAssembly Binary Toolkit (WABT) [22] provides tools for converting between `wasm` and `wat` representations, as well as for the de-compilation and validation of WebAssembly binaries.

Use Cases. WebAssembly has been adopted for various applications on the web due to its near-native execution performance, such as data compression, game engines, and natural language processing. However, the usage of WebAssembly is not only limited to the web. It is also being extended to desktop applications [6], mobile devices [7], cloud computing [8], IoT [12][13], and embedded devices [14].

2. Security

Environment. WebAssembly modules run in a sandboxed environment which uses fault isolation techniques to separate it from the host environment. As a result of this, modules have to go through APIs to access external resources. For instance, modules that run in the web browser must use JavaScript APIs to interact with the Document Object Model (DOM). Similarly, stand-alone runtimes must use APIs, like WASI, to access system resources like files. In addition to this, modules must adhere to the security policies implemented by its host environment, such as the Same Origin Policy (SOP) [23] enforced by web browsers, which restricts the flow of information between web pages from different origins.

Memory. Unlike native binaries, which have access to the entire memory space allocated to the process, WebAssembly modules only have access to a contiguous region of memory known as linear memory. This memory is untyped and byte-addressable, and its size is determined by the data present in the binary. The size of linear memory is a multiple of a WebAssembly page, each being 64 KiB in size. When a WebAssembly module is instantiated, it uses the appropriate API call to allocate the memory that is needed for its execution. The host environment then creates a managed buffer, typically an `ArrayBuffer`, to store the linear memory. This means that the WebAssembly module accesses the physical memory indirectly through the managed buffer, which ensures that it can only read and write data within a limited area of the memory.

Control Flow Integrity. WebAssembly enforces structured control flow, organizing instructions into well-nested blocks within functions. It restricts branches to the end of surrounding blocks or within the current function, with multi-way branches targeting only pre-defined blocks. This prevents unrestricted go-tos or executing data as bytecode, eliminating attacks like shellcode injection or the misuse of indirect jumps. Additionally, execution semantics ensure safety for direct function calls through explicit indexing and protected returns with a call stack. Indirect function calls undergo runtime checks for type

signatures, establishing coarse-grained, type-based control-flow integrity. Additionally, the LLVM compiler infrastructure has been adapted to include a fine-grained control flow integrity feature, specifically designed to support WebAssembly [24].

3. Vulnerabilities

Inherent vulnerabilities in the source code can lead to subsequent vulnerabilities in WebAssembly modules [25]. Specifically, buffer overflows in memory-unsafe languages like C and C++ can overwrite constant data or the heap in WebAssembly modules. Despite WebAssembly's sandboxing, these vulnerabilities allow malicious script injection into the module's data section, which is accessible via JavaScript APIs. An example of this is the Emscripten API [20], which allows developers to access data from WebAssembly modules and inject these into the DOM, which can lead to Cross Site Scripting (XSS) attacks [26]. Notably, two-thirds of WebAssembly binaries are compiled from memory-unsafe languages [27], and these attacks have been shown to be practical in real-world scenarios [25]. For instance, Fastly, a cloud platform that offers edge computing services, experienced a 45 min disruption on 8 June 2021, when a WebAssembly binary with a vulnerability was deployed [28]

4. Smart Contracts

Smart contracts are computer programs that are stored on a blockchain, designed to automatically execute once predetermined conditions are met, eliminating the need for intermediaries. As initially proposed by Nick Szabo in 1994 [29], long before the advent of Bitcoin, they have since gained widespread popularity alongside the rise of blockchain technology and cryptocurrencies. The inherent properties of blockchain, such as transparency, security, and immutability, make smart contracts particularly appealing for cryptocurrency transactions. This ensures that once the terms of the contract are agreed upon and coded into the blockchain, they can be executed without the possibility of fraud or third-party interference. Smart contracts can facilitate a variety of transactions, from the transfer of cryptocurrency between parties to the automation of complex processes in finance, real estate, and beyond. Due to its near-native performance, WebAssembly has been adopted by blockchain platforms, such as EOSIO [10] and NEAR [11], as their smart contract runtime. Ethereum has included WebAssembly in the roadmap for Ethereum 2.0, positioning it as the successor to the Ethereum Virtual Machine (EVM) [9].

However, as with any technology, smart contracts are not without their challenges and vulnerabilities. The immutable nature of blockchain means that once a smart contract is deployed, it cannot be modified, making the correction of vulnerabilities in its code challenging. Several incidents have highlighted the potential financial and security risks associated with vulnerabilities in WebAssembly smart contracts. For instance, random number generation vulnerabilities led to the theft of approximately 170,000 EOS tokens [30]. Similarly, the fake EOS transfer vulnerability in the EOSCast smart contract has led to the theft of approximately 60,000 EOS tokens [31]. The forged transfer notification vulnerability in EOSBet has resulted in the loss of 140,000 EOS tokens [31]. Based on the average price of EOS tokens at the time of the attacks, the combined financial impact of these three vulnerabilities amounted to roughly USD 1.9 million. Additionally, around 25% of WebAssembly smart contracts have been found to be vulnerable [32].

5. Cryptojacking

Cryptojacking, also known as drive-by mining, involves using a website visitor's hardware resources for mining cryptocurrencies without their consent. Previously, cryptojacking was implemented using JavaScript. However, in recent years WebAssembly has been utilized due to its computational efficiency. The year after WebAssembly was released, there was a 459% increase in cryptojacking [33]. The following year, researchers found that over 50% of all sites using WebAssembly were using it for cryptojacking [34]. To counter this trend, researchers developed several static and dynamic detection methods for identifying WebAssembly-based cryptojacking.

While there are theories suggesting that WebAssembly can be used for other malicious purposes, like tech support scams, browser exploits, and script-based keyloggers [35], evidence of such misuse in real-world scenarios has not been documented. As a result, there are no analysis techniques for detecting such malicious WebAssembly binaries. Consequently, discussions about malicious WebAssembly binaries in this research mainly refer to crypto mining binaries.

References

1. w3Techs. Usage Statistics of JavaScript as Client-Side Programming Language on Websites. December 2022. Available online: https://w3techs.com/technologies/details/cp-javascript (accessed on 4 November 2022).
2. Contributors to Wikimedia Projects. ActiveX-Wikipedia. 2022. Available online: https://en.wikipedia.org/wiki/ActiveX (accessed on 4 November 2022).
3. Google. Native Client-Chrome Developers. 2021. Available online: https://developer.chrome.com/docs/native-client (accessed on 4 November 2022).
4. mdn web docs. asm.js-Game Development | MDN. 2022. Available online: https://developer.mozilla.org/en-US/docs/Games/Tools/asm.js?source=post_page (accessed on 4 November 2022).
5. Can I Use WebAssembly | Can I Use… Support Tables for HTML5, CSS3, etc. 2022. Available online: https://caniuse.com/wasm (accessed on 12 November 2022).
6. Møller, A. Technical perspective: WebAssembly: A quiet revolution of the Web. Commun. ACM 2018, 61, 106.
7. Pop, V.A.B.; Virtanen, S.; Sainio, P.; Niemi, A. Secure Migration of WebAssembly-Based Mobile Agents between Secure Enclaves. Master's Thesis, University of Turku, Turku, Finland, 2021.
8. Fastly. Fastly Docs. 2022. Available online: https://docs.fastly.com/products/compute-at-edge (accessed on 23 November 2022).
9. Ewasm. Ethereum WebAssembly (Ewasm)-Ethereum WebAssembly. 2021. Available online: https://ewasm.readthedocs.io/en/mkdocs (accessed on 7 November 2022).
10. Eosio. EOS Virtual Machine: A High-Performance Blockchain WebAssembly Interpreter–EOSIO. 2021. Available online: https://eos.io/news/eos-virtual-machine-a-high-performance-blockchain-webassembly-interpreter (accessed on 9 November 2022).
11. NEARWhat Is a Smart Contract?|NEAR Documentation. 2022. Available online: https://docs.near.org/develop/contracts/whatisacontract (accessed on 11 November 2022).
12. Liu, R.; Garcia, L.; Srivastava, M. Aerogel: Lightweight Access Control Framework for WebAssembly-Based Bare-Metal IoT Devices. In Proceedings of the 2021 IEEE/ACM Symposium on Edge Computing (SEC), San Jose, CA, USA, 14–17 December 2021; IEEE: Piscataway, NJ, USA, 2021; pp. 94–105.
13. Mäkitalo, N.; Mikkonen, T.; Pautasso, C.; Bankowski, V.; Daubaris, P.; Mikkola, R.; Beletski, O. WebAssembly modules as lightweight containers for liquid IoT applications. In Proceedings of the International Conference on Web Engineering, Biarritz, France, 18–21 May 2021; Springer: Cham, Switzerland, 2021; pp. 328–336.
14. Scheidl, F. Valent-Blocks: Scalable high-performance compilation of WebAssembly bytecode for embedded systems. In Proceedings of the 2020 International Conference on Computing, Electronics & Communications Engineering (iCCECE), Southend, UK, 17–18 August 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 119–124.
15. Haas, A.; Rossberg, A.; Schuff, D.L.; Titzer, B.L.; Holman, M.; Gohman, D.; Wagner, L.; Zakai, A.; Bastien, J. Bringing the web up to speed with WebAssembly. In Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, Barcelona, Spain, 18–23 June 2017; pp. 185–200.
16. w3c. World Wide Web Consortium (W3C) Brings a New Language to the Web as WebAssembly Becomes a W3C Recommendation. 2019. Available online: https://www.w3.org/2019/12/pressrelease-wasm-rec.html.en (accessed on 16 November 2022).
17. Mozilla Using the WebAssembly JavaScript API-WebAssembly | MDN. 2022. Available online: https://developer.mozilla.org/en-US/docs/WebAssembly/Using_the_JavaScript_API (accessed on 29 November 2022).
18. Node.js. Node.js. 2022. Available online: https://nodejs.org/en (accessed on 27 November 2022).
19. Wasi. WASI|. 2022. Available online: https://wasi.dev (accessed on 25 November 2022).

20. Emscripten. Main—Emscripten 3.1.26-git (dev) Documentation. 2022. Available online: https://emscripten.org (accessed on 1 December 2022).
21. Rustwasm. Wasm-Pack. 2022. Available online: https://rustwasm.github.io/wasm-pack (accessed on 23 November 2022).
22. WebAssembly. Wabt. 2022. Available online: https://github.com/WebAssembly/wabt (accessed on 26 November 2022).
23. w3c. Same Origin Policy-Web Security. 2022. Available online: https://www.w3.org/Security/wiki/Same_Origin_Policy (accessed on 26 November 2022).
24. Docs, W. Security-WebAssembly. 2022. Available online: https://webassembly.org/docs/security/#users (accessed on 3 November 2022).
25. Lehmann, D.; Kinder, J.; Pradel, M. Everything Old is New Again: Binary Security of WebAssembly. In Proceedings of the 29th USENIX Security Symposium (USENIX Security 20), 12–14 August 2020; USENIX Association: Berkeley, CA, USA, 2020; pp. 217–234.
26. McFadden, B.; Lukasiewicz, T.; Dileo, J.; Engler, J. Security Chasms of Wasm. 2018. NCC Group Whitepaper. Available online: https://git.edik.cn/book/awesome-wasm-zh/raw/commit/e046f91804fb5deb95affb52d6348de92c5bd99c/spec/us-18-Lukasiewicz-WebAssembly-A-New-World-of-Native_Exploits-On-The-Web-wp.pdf (accessed on 5 January 2024).
27. Hilbig, A.; Lehmann, D.; Pradel, M. An Empirical Study of Real-World WebAssembly Binaries: Security, Languages, Use Cases. In Proceedings of the Web Conference 2021, Ljubljana, Slovenia, 19–23 April 2021; WWW '21. pp. 2696–2708.
28. Fastly. Summary of June 8 Outage. 2021. Available online: https://www.fastly.com/blog/summary-of-june-8-outage (accessed on 29 November 2022).
29. Szabo, N. Smart Contracts. 1994. Available online: https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smar (accessed on 5 February 2024).
30. PeckShield. Defeating EOS Gambling Games: The Tech Behind Random Number Loophole. Medium 2018. Available online: https://peckshield.medium.com/defeating-eos-gambling-games-the-tech-behind-random-number-loophole-cf701c616dc0 (accessed on 5 January 2024).
31. Huang, Y.; Jiang, B.; Chan, W.K. EOSFuzzer: Fuzzing EOSIO Smart Contracts for Vulnerability Detection. In Proceedings of the 12th Asia-Pacific Symposium on Internetware, Singapore, 1–3 November 2020; ACM: New York, NY, USA, 2020.
32. He, N.; Zhang, R.; Wang, H.; Wu, L.; Luo, X.; Guo, Y.; Yu, T.; Jiang, X. EOSAFE: Security Analysis of EOSIO Smart Contracts. In Proceedings of the 30th USENIX Security Symposium (USENIX Security 21), Vancouver, BC, Canada, 11–13 August 2021; USENIX Association: Berkeley, CA, USA, 2021; pp. 1271–1288.
33. Alliance, C.T. The Illicit Cryptocurrency Mining Threat. 2018. Available online: https://cyberthreatalliance.org/wp-content/uploads/2018/09/CTA-Illicit-CryptoMining-Whitepaper.pdf (accessed on 25 November 2022).
34. Musch, M.; Wressnegger, C.; Johns, M.; Rieck, K. New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild. In Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Gothenburg, Sweden, 19–20 June 2019; Springer: Cham, Switzerland, 2019; pp. 23–42.
35. Lonkar, A.; Chandrayan, S. The dark side of WebAssembly. In Proceedings of the Virus Bulletin Conference, Montreal, QC, Canada, 3–5 October 2018.