Security analysis of standalone implementations of the WebAssembly memory model

Supervisor

Quentin MICHAUD, PhD CIFRE student at Thales and Télécom SudParis[1]

Contact: `quentin.michaud@telecom-sudparis.eu`

Description

WebAssembly [1], [2], or Wasm for short, has been created as a fast and secure-by-design answer to the always increasing need for complex computation in browsers, such as 3D rendering, 3D model parsing, gaming, hardware emulation or physics workloads (e.g. computational fluid dynamics) [3]. The success of WebAssembly as a portable Instruction Set Architecture (ISA) and binary format has prompted its adoption in many applications besides browsers. Today, we can find WebAssembly in smart contracts, embedded devices [4], in secure plugins [5], [6], in Function as a Service (FaaS) platforms [7], or as a standalone runtime [8]. The latter has a huge impact on the cloud world and the computing world in general. Some see in the flexibility of WebAssembly a universal binary format that could be distributed seamlessly across operating systems and hardware architectures. It also appears in various cloud-related projects and is considered as an alternative to Linux-based containers [9], promising to be more portable, lightweight and secure.

WebAssembly claims strong security. By default, it provides sandboxing between different WebAssembly instances and between WebAssembly and its host. It also enforces control-flow integrity, and protection against code reuse attacks. However, the security of WebAssembly has been challenged in several works [10], [11]. First, WebAssembly offers weak protection against memory corruption attacks compared to native binaries. Some vulnerabilities, such as stack-based buffer overflows, have been present in native binaries for a long time, but are mitigated with mechanisms such as Stack Smashing Protection (SSP). This protection was initially absent in WebAssembly. Second, differences in design between WebAssembly and native binaries make the former vulnerable to attacks that are not possible in native binaries. One example is the corruption of heap data using a stack-based buffer overflow.

Goals

The security of WebAssembly, and of standalone (WASI) WebAssembly in particular, is yet to have been studied in depth. One research question is if the current runtime implementations garantee that the WebAssembly specification detailed in the specification are correctly translated in the implementations. The investigation will focus on understanding where the different values from the different memories in the WebAssembly virtual machine are stored on the host machine, and if this layout is potentially vulnerable to memory vulnerabilities from within WebAssembly.

A proposition of steps for conducting the project would be:

1. Familiarization with the WebAssembly specification, its memory model, the WASI APIs, existing tooling (compilers, libraries, …) and previous academic work on WebAssembly security.
2. Define precisely the goal and the scope of the analysis (which memories are studied, what are the expectations on the implementations, what could be the potential vulnerabilities and their impact, …)

[1] https://www.theses.fr/s374883

3. Design an experimentation protocol that will allow to automatically assess the points that have been defined in the previous step. This protocol must be working across various compilation toolchains and runtimes.
4. Implement a tool that operate the experimentation protocol and run it on target runtimes and toolchains.
5. Collect the results, propose interpretations, and possible remediations.

Bibliography

[1] A. Haas *et al.*, "Bringing the web up to speed with WebAssembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, in PLDI 2017. New York, NY, USA: Association for Computing Machinery, Jun. 2017, pp. 185–200. doi: 10.1145/3062341.3062363.

[2] A. Rossberg, "WebAssembly Core Specification," Dec. 2019. [Online]. Available: https://www.w3.org/TR/wasm-core-1/

[3] M. Sakuta, "Computational Fluid Dynamics simulation with Webassembly and Rust." Accessed: Apr. 17, 2024. [Online]. Available: https://github.com/msakuta/cfd-wasm

[4] R. Gurdeep Singh and C. Scholliers, "WARDuino: a dynamic WebAssembly virtual machine for programming microcontrollers," in *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, Athens Greece: ACM, Oct. 2019, pp. 27–36. doi: 10.1145/3357390.3361029.

[5] "Extism - make all software programmable. Extend from within. | Extism - make all software programmable. Extend from within.." Accessed: Apr. 17, 2024. [Online]. Available: https://extism.org/

[6] S. Narayan, T. Garfinkel, S. Lerner, H. Shacham, and D. Stefan, "Gobi: WebAssembly as a Practical Path to Library Sandboxing." Accessed: Apr. 17, 2024. [Online]. Available: http://arxiv.org/abs/1912.02285

[7] V. Kjorveziroski and S. Filiposka, "WebAssembly Orchestration in the Context of Serverless Computing," *Journal of Network and Systems Management*, vol. 31, no. 3, p. 62–63, Jul. 2023, doi: 10.1007/s10922-023-09753-0.

[8] L. Clark, "Standardizing WASI: A system interface to run WebAssembly outside the web – Mozilla Hacks - the Web developer blog." Accessed: Apr. 17, 2024. [Online]. Available: https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface

[9] "wasmCloud." Accessed: Apr. 17, 2024. [Online]. Available: https://wasmcloud.com/

[10] D. Lehmann, J. Kinder, and M. Pradel, "Everything Old is New Again: Binary Security of WebAssembly," 2020, pp. 217–234. Accessed: Sep. 14, 2023. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann

[11] Q. Stiévenart, C. De Roover, and M. Ghafari, "Security risks of porting C programs to webassembly," in *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, in SAC '22. New York, NY, USA: Association for Computing Machinery, May 2022, pp. 1713–1722. doi: 10.1145/3477314.3507308.