POSTER: Leveraging eBPF to enhance sandboxing of WebAssembly runtimes

Marco Abbadini
marco.abbadini@unibg.it
Università degli Studi di Bergamo
Bergamo, Italy

Michele Beretta
michele.beretta@unibg.it
Università degli Studi di Bergamo
Bergamo, Italy

Dario Facchinetti
dario.facchinetti@unibg.it
Università degli Studi di Bergamo
Bergamo, Italy

Gianluca Oldani
gianluca.oldani@unibg.it
Università degli Studi di Bergamo
Bergamo, Italy

Matthew Rossi
matthew.rossi@unibg.it
Università degli Studi di Bergamo
Bergamo, Italy

Stefano Paraboschi
stefano.paraboschi@unibg.it
Università degli Studi di Bergamo
Bergamo, Italy

ABSTRACT

WebAssembly is a binary instruction format designed as a portable compilation target enabling the deployment of untrusted code in a safe and efficient manner. While it was originally designed to be run inside web browsers, modern runtimes like Wasmtime and WasmEdge can execute WebAssembly directly on various systems. In order to access system resources with a universal hostcall interface, a standardization effort named WebAssembly System Interface (WASI) is currently undergoing. With specific regard to the file system, runtimes must prevent hostcalls to access arbitrary locations, thus they introduce security checks to only permit access to a predefined list of directories. This approach not only suffers from poor granularity, it is also error-prone and has led to several security issues. In this work we replace the security checks in hostcall wrappers with eBPF programs, enabling the introduction of fine-grained per-module policies. Preliminary experiments confirm that our approach introduces limited overhead to existing runtimes.

CCS CONCEPTS

• **Security and privacy → Software and application security**; **Access control**; *File system security*.

KEYWORDS

Sandboxing, Access control, WebAssembly runtime, eBPF

ACM Reference Format:
Marco Abbadini, Michele Beretta, Dario Facchinetti, Gianluca Oldani, Matthew Rossi, and Stefano Paraboschi. 2023. POSTER: Leveraging eBPF to enhance sandboxing of WebAssembly runtimes. In *ACM ASIA Conference on Computer and Communications Security (ASIA CCS '23), July 10–14, 2023, Melbourne, VIC, Australia.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3579856.3592831

ASIA CCS '23, July 10–14, 2023, Melbourne, VIC, Australia

ACM ISBN 979-8-4007-0098-9/23/07.
https://doi.org/10.1145/3579856.3592831

1 INTRODUCTION

WebAssembly (Wasm) [15] is a popular binary instruction format that enables the execution of untrusted code in a safe, isolated environment. Moreover, it is a portable compilation target for different languages, and can be executed efficiently on a wide range of platforms without the need of dedicated hardware. Wasm was originally meant to be run inside web browsers, but given the considerable advantages it brings, many runtimes that allow execution in standalone mode have been developed recently. Popular examples are Wasmtime, WasmEdge, Wasmer, and WAMR.

To answer the developers' need to access resources of the host system from within the runtime, a standardization effort called WebAssembly System Interface (WASI) [28] is undergoing. Its goal is to provide a stable and multi-platform system interface. To be WASI-compliant, each runtime must implement all the calls defined in the interface with dedicated functions, which are named hostcalls. However, implementing these functions is non-trivial, since (i) the code must not introduce violations to the Wasm memory model, and (ii) it is possible to break the separation between the system and the isolated environment in which the Wasm module is executed. The solution adopted by current runtimes leverages WASI Libc [27], a library providing POSIX-compatible APIs built on top of hostcalls.

Currently, every WASI-compliant runtime implements the proposed file system interface with a libpreopen-like layer [21]. Whenever the runtime receives a request to open a file, it first checks whether the path belongs to the authorized list of directories, then it opens the file on behalf of the Wasm program, redirecting the content to the caller. Previous work [7, 16, 18] proved the approach to be error-prone, leaving the system unprotected when a vulnerability was introduced in a hostcall wrapper (Figure 1). Moreover, this approach provides limited flexibility, as it is associated with directory-based granularity instead of file-based. Lastly, in order to audit the policy regulating resource access, one must find the permissions by looking at the code. We claim that there is no practical advantage in having several implementations of the same access control checks for different runtimes. Our idea is to replace the user-space runtime-specific security checks with a single in-kernel implementation that leverages eBPF [26]. There are considerable advantages in doing so: (i) it permits to decouple the implementation of hostcall wrappers and the access control details, minimizing the risk of bugs [3, 17, 25], (ii) it enables the introduction of per-module policies with file-based granularity, and (iii) it fulfills

Figure 1: Current implementation of WASI by runtimes. A bug present in a hostcall wrapper permits the module to read the unauthorized directory on the left (red dotted arrow)

Wasm's promise of portability as eBPF programs are portable across di erent kernel versions [22] and also operating systems, thanks to Microsoft's undergoing e ort to port eBPF to Windows [20].

2 THREAT MODEL

Our assumptions re ect the threat model employed by Wasm runtimes. We assume that the code executed by the runtime is either untrusted or it is trusted but potentially a ected by security vulnerabilities due to bugs. The goal of the attacker providing the code is to bypass the security checks enforced by the runtime to get access to the host le system. To ful ll this objective, the attacker can leverage the interface provided by WASI and send any argument. Runtime escapes caused by memory corruption or alteration of the program ow are out of scope of our work, since protection can be provided by other existing solutions (e.g., [7]).

3 ARCHITECTURE

Our analysis starts from the scenario illustrated in Figure 1. Currently, WASI-compliant runtimes implement dedicated user-space wrappers to enforce the security boundaries of hostcalls. File system access is granted by the user on a set of pre-opened directories that are speci ed via CLI before the Wasm module is run (e.g., with the `--dir` option). We follow a similar approach, asking the user to state the permissions of each Wasm module in a JSON policy le. Contrary to existing runtimes, permissions can be granted with le-based granularity. Three permissions are available: `read` (i) to open and read a le, (ii) `write` to modify, truncate and append content to a le, and (iii) `delete` to remove the le. When permissions are related to a directory, `read` translates to listing its content, `write` allows to create and delete les within it. We extended the Wasmtime and WasmEdge runtimes to load the policy at startup, and, instead of pre-opening the directories available to the Wasm module, we enforce the policy with eBPF. eBPF code is split into programs attached to a kernel- or user-space function called *hook point* and executed whenever the hook is reached. Programs have visibility of function parameters, they can persist state and share it with user space using *maps*, and most of all they can enforce security checks based on this information. Once the policy is encoded inside the map and the eBPF programs are loaded, the runtime instantiates the Wasm module selected by the user (arrow Ⓐ in Figure 2). At this stage, the modi ed runtime invokes a dedicated user probe specifying as a parameter the policy that con nes the loaded Wasm module (Ⓑ). The argument is captured by a dedicated eBPF program

Figure 2: Work ow of our proposal. The runtime instantiates the Wasm module (Ⓐ), and con gures the associated policy calling the traced user probe (Ⓑ). After the Wasm module is run, all the hostcalls issued by the program (①) are restricted by eBPF (②, ③)

that also annotates the identi er of the thread running the Wasm interpreter in a tracing map. We highlight that the policy is activated before the runtime executes the module (i.e., before untrusted code is interpreted). The consequence is that, from this point on, all the hostcalls performed by the Wasm module are restricted by our eBPF programs (arrow ①, ②). The eBPF programs that make the security decisions are evaluated every time a le-related kernel security hook is reached (e.g., `security_file_open`), and any access decision is enforced at kernel level. When an unauthorized request is performed by the Wasm code (③), the related eBPF program detects the violation and denies the request, returning to the caller a permission denied error. When the execution of untrusted Wasm code terminates, another eBPF program is responsible for removing the access restriction from the thread executing the Wasm runtime. No further intervention from the runtime is required, as the maps and the eBPF programs are automatically removed from the kernel immediately after the process running the runtime terminates.

This architecture o ers several advantages. First, it eliminates the risks coming from buggy user-space security checks (e.g., wrong le path resolution [19], wrong directory removal [8]). Then, by leveraging kernel hook points [26], our approach allows the runtime developer to focus on the interaction between Wasm code and the memory unsafe system call, leaving aside authorizations and policy-related issues. Lastly, access constraints can be audited by simply looking at the JSON policy, instead of inspecting the code.

4 EXPERIMENTS

To investigate the overhead introduced by our solution we implemented it in WasmEdge and Wasmtime, two industrial state-of-the-art Wasm runtimes. The evaluation has been performed in the following test environment: an Ubuntu 22.04 LTS server powered by an AMD Ryzen 2950X CPU with 16 cores, 128 GB RAM, and 2 TB SSD. In order to assess the performance, we tested one of the most popular binaries that can be compiled to Wasm with support to WASI: `uutils coreutils`, the porting of the coreutils in Rust [9]. First, we compiled the coreutils with the `wasm32-wasi` target, and applied runtime-speci c optimization (with `wasmedge` [2]