State of Security in WebAssembly Binaries

Farhan Saif
Dept. of Computer Science
Islamic University Of Technology
Gazipur, Bangladesh
farhansaif@iut-dhaka.edu

Adib Abrar Kabeer
Dept. of Computer Science
Islamic University Of Technology
Gazipur, Bangladesh
adibabrar@iut-dhaka.edu

Shihab Sikder
Dept. of Computer Science
Islamic University Of Technology
Gazipur, Bangladesh
shihabsikder@iut-dhaka.edu

***Abstract*—Finding vulnerabilities in binaries and exploiting them is a well researched field in computer security. With the advent of new byte code format for web - WebAssembly, classic techniques as well as new and innovative exploitation possibilities are arising as a new field of study. In this paper, we discuss about the prospects of vulnerabilities in WebAssembly binaries and their possible mitigation as proposed in existing studies.**

I. Introduction

WebAssembly, or Wasm, is a byte code format which is designed to run directly on web browsers. It was released in 2017 and its goal is to run high performance codes in browsers. Despite being so young, it's gaining traction in the web development community quickly. By analyzing Alexa's most visited sites, it was found that, 1 in 600 sites of top one million most visited websites uses Wasm modules [**?**]. With its increasing use, possible security risks in Wasm binaries are also surfacing, which will require deep analysis in order to ensure the practicality of Wasm binaries in modern web development.

II. WebAssembly

A. Overview

WebAssembly is designed to be the byte code format in web. It is designed to be a portable compilation target for programming languages, enabling its use in web client and server applications [**?**]. As of 2021, WebAssembly is supported in 91.87% browsers including 4 major browser engines, i.e. Mozilla Firefox, Google Chrome, Safari and Microsoft Edge [**?**].

B. Background

JavaScript is the primary language of web browsers. Most modern web frameworks compile or transpile into JavaScript so that the applications can run on the browser. However, there has been ongoing research for a new bytecode format in browsers for a long time. The goal was to have a safe, fast, portable and compact language that can act as the compilation target for high level languages. Microsoft's ActiveX, NaCl (Native Client) [**?**], Emscripten [**?**] and many other technologies have been developed in order to attempt this feat but they were not good enough to replace the old and renowned JavaScript . In 2017, the first version of WebAssembly was released and it proved itself to be the most complete browser byte code format in existence [**?**].

C. Compilation

Wasm is a 32-bit machine language which is designed to be compiled primarily from C, C++ and Rust [**?**]. The C/C++/Rust source file is compiled into Wasm binaries directly, but an intermediate assembly-like syntax can be extracted, which is known as a *wat* (WebAssembly Text) file [**?**].

D. Syntax

Fig. 1. A simple Wasm module (in *wat* format) that adds two 32-bit integers (*i32*) and returns the sum.

Data Types: Unlike other bytecode languages, Wasm supports 4 type primitives - *i32* (32-bit integer), *i64* (64-bit integer), *f32* (32-bit float) and *f64* (64-bit float).

Control Flow: WebAssembly uses structured control flow where function instructions are nested in blocks. Branches can transfer the flow to the end of these blocks only. A separate indirect call method is used for function pointers, which controls program flow using indices from a function table.

E. Memory

Storage in Wasm is designed as a global single array of bytes, i.e. simple linear memory. Memory address pointers are in *i32* data type, as the addresses are 32-bit. Heap and stack are both implemented onto the linear memory. Wasm allocates memory on its own and does not provide memory management.

F. Execution Environment

Wasm modules are executed in a host environment like web browsers or NodeJS. They provide the Wasm modules with necessary APIs, e.g. browser API. These environments provide a sandbox environment for the Wasm modules, so that unsafe activities can be prevented.

III. Memory Corruption Vulnerabilities

Memory corruption vulnerabilities include a set of primitives that enables attackers to overwrite program memory causing unpredictable and malicious behaviour. These are

the most common vulnerabilities found in memory unsafe languages. There is a possibility of these vulnerabilities translating into Wasm binaries when compiled. After decades of hardening, possible memory corruptions in x86 binaries are well defined with proper mitigation. The same cannot be said for WebAssembly. Possible memory corruption vulnerabilities in WebAssembly can be [**?**]:

A. Stack-based Buffer Overflow

WebAssembly dynamically allocates memory and does not provide any manual memory management. Even though Wasm VM isolates Wasm module memory access, parts of C/C++ function data are stored on unmanaged stack of the VM. So, stack-based buffer overflow vulnerabilities are prevented in internal memory but not the linear unmanaged memory of Wasm sandbox. Native platforms can prevent this type of exploitations using stack canaries.

B. Stack Overflow

Wasm modules can increase their memory allocation with certain API calls. That means, it is possible to control the memory allocation size and insert corrupted input data into the stack to invoke stack overflow. Native platforms use guard page to prevent this type of attacks but such mitigation is absent in WebAssembly.

C. Heap Metadata Corruption

Wasm developers can choose their preferred memory allocator as per their intended use. Default Wasm allocators "dlmalloc" is hardened against heap metadata corruption attacks. However, since the binary size is an important consideration in web development, developers may use lightweight allocators which might not be hardened against memory corruption. In this case, if there is no fortification, attackers can write to adjacent metadata of chunks in heap memory when these allocators allocate/deallocate memory.

D. Overwriting Stack Data

The unmanaged stack in WebAssembly execution memory contains function-scoped data. With a proper write primitive, it is possible to overwrite function-scoped local data in the stack.

E. Overwriting Heap Data

Linear stack based overflow can easily overwrite heap data as heap and stack share the same linear memory in WebAssembly. Native mitigation like guard pages are absent in WebAssembly, which means there is no way to avoid overflow based vulnerabilities in the linear memory.

F. Injecting Code into Host

As mentioned before, WebAssembly modules use different API calls from its host environment in order to extend its uses to practical environment. Using functions like eval/exec, found in browser/NodeJS host environment, Wasm modules can be developed to execute code in the host environment. As a result, host environment vulnerabilities can be exploited from within Wasm modules. The arrays of possible exploitation include remote code execution, cross site scripting etc.

A detailed study on Wasm design and specifications [**?**] indicates that these old vulnerabilities found in the high level programming languages that can compile into Wasm, may translate into equivalent vulnerabilities in Wasm binaries. Though there are certainly more fortification in WebAssembly by default, security is not completely guaranteed.

IV. Side Channel Attacks

Side channel attacks target the execution environment of the system by exploiting indirect effects of a system or its hardware. An example of this is Spectre [**?**], which affects modern microprocessors. Recent processors use branch predictions to increase performance and throughput. On these processors, execution resulting from a branch misprediction may leave side effects that can reveal private data. JavaScript Just-In-Time (JIT) compilers and transpilers are directly affected by this vulnerability.

Wasm isolates untrusted modules using run time as well as compile time checks. This includes heap memory access and indirect function call checks. The validity of return addresses are also ensured using a safe stack. However, these fortifications can be bypassed using the following techniques [**?**]:

A. Spectre-PHT (Pattern History Table)

The pattern history table (PHT) can be exploited to confuse the conditional branch predictor and make it mispredict a path. A wrong path execution like this can be exploited to bypass control flow and memory isolation.

B. Spectre-BTB (Branch Target Buffer)

BTB helps in predicting the target address of indirect jump instructions in programs. Similar to PHT, BTB's entries can be changed maliciously to change control flow to a specific target.

C. Spectre-RSB (Return Stack Buffer)

RSB stores the return addresses of executed call instructions and helps in predicting the return points from executed functions. By overflowing RSB using a series of call and ret instructions, control flow of the program can be hijacked.

D. Sandbox Breakout

Wasm is widely used in FaaS (Function as a Service) platforms, where Spectre-PHT or BTB can be used to access data/control outside the sandbox. This is known as Sandbox breakout.

E. Sandbox Poisoning

After using Spectre attack to misdirect the control flow, sandbox data can be leaked by accessing data from sandbox cache or similar state stores.

F. Host Poisoning

The host runtime can also be exploited in a similar way as sandbox, and host system data can be accessed maliciously.

V. Proposed Mitigation

While finding vulnerabilities in WebAssembly, studies have proposed [**?**] possible mitigation for different types of attacks. For memory corruptions, the following mitigation can be directly incorporated into WebAssembly standard:

A. Multi Memory Implementation

Rather than having a single linear memory, multi memory system in WebAssembly [**?**] can enable the system to have separate data spaces for heap, stack and constant data. As a result, indirect overflows and pointer forging can be prevented for the most part. This is a proposed specification for WebAssembly and may be implemented into the language in the future.

B. MS-Wasm Proposal

The MS-Wasm Proposal [**?**] suggests the addition of memory segments of specific size and lifetime in the WebAssembly language. Implementing this may prove to be hard for hosts but it provides high memory safety.

C. Address Space Layout Randomization

Using the presently available linear memory with randomized address layout, an additional layer of security can provided against memory based exploits. This causes obfuscation of memory locations of contiguous data segments in the program.

D. Safe Unlinking

Safe unlinking may prevent metadata corruption as it disables exploits from writing into arbitrary chunks in memory.

E. Guard page

Guard Page protection mechanism triggers page fault when the stack grows into restricted guarded pages. If such page fault occurs during any exploitation, the program will simply crash and invalid data access can be prevented.

To address side channel attacks like Spectre, Swivel [**?**] has been developed to mitigate exploitation:

F. Swivel-SFI

Swivel-SFI mitigates sandbox breakout, sandbox poisoning and host poisoning through a series of fortifications. These fortifications include the use of a separate stack to protect return addresses, BTB flushing to prevent polluting BTB entries and elimination of CBP poisoning.

G. Swivel-CET

Swivel-CET mitigates sandbox breakout and poisoning with shadow stack, forward-edge CFI and conditional BTB flushing. It also includes register interlocking and leak prevention during execution to provide poisoning detection and fortification. This, even if the vulnerabilities are exploited, the defenses can protect data from getting leaked.

VI. Mitigation Challenges

The biggest challenge in the face of WebAssembly security updates is browser support. There is a large number of browsers in the market and all of them need to support Wasm in order for it to become the staple byte code for web. Thus, WebAssembly specification update means that the maintainer of all these browsers will need to update their VMs accordingly to match the new security standard. This method has no alternative and as a result it will take some time for Wasm to mature into what it aims to become.

VII. Analysis of WebAssembly

WebAssembly security requires continuous analysis of Wasm modules and programs to develop further. Native application domain already has a huge number of tools to have their binaries analyzed for security concerns. Since WebAssembly is a new language, there is a scarcity of tools in this field. However, many high quality tools have been developed already. Some of these tools are:

A. Wasabi

Wasabi [**?**] is a framework for dynamic analysis of WebAssembly binaries. This open source framework instruments Wasm binary while preserving program behaviour and affecting performance and size slightly. A security researcher can use Wasabi to write general-purpose custom dynamic analyses, e.g. instruction count, call graph extraction and analysis, memory tracing and taint analysis.

B. WasmBench

WasmBench [**?**] is the largest open-source Wasm binary database. It gathered real WebAssembly programs from existing websites using web crawling, GitHub repositories and manual module extraction. For any binary analysis toolset, a dataset of real binaries is required for experimentation and validation. WasmBench can fulfill that role for future binary analysis tools targeting Wasm.

C. Fuzzm

Fuzzm [**?**] is a greybox fuzzer for WebAssembly binaries. It integrates an AFL style fuzzer to test inputs on Wasm binaries. The main goal of this fuzzer is to identify suspicious program behaviour, which is not present in WebAssembly toolchain by default. Fuzzm provides hardening against stack and heap based attacks with the use of canaries. As it is the only fuzzer available for WebAssembly at present, Fuzzm

TABLE I
WEBASSEMBLY BINARY TYPE COUNT IN 100 RANDOMLY SELECTED SAMPLES FROM WASMBENCH

Types	Count
Games	25
Text Processing	11
Visualization	11
Media Processing	9
Demo	7
Wasm Test	5
Chat	3
Online Gambling	2
Barcode & QR code scanning	2
Room Planning & Furniture	2
Blogging	2
Crypto-currency wallet	2
Regular Expressions	1
Hashing	1
PDF Viewer	1

is pioneering in the development of binary fuzzer targeting Wasm.

VIII. CONCLUSION

WebAssembly developers originally claimed Wasm to be completely safe [**?**] in 2017. However, several studies have proven that Wasm is not as safe [**?**] [**?**] as the specifications describe it to be. Wasm binaries inherit some vulnerabilities from their source code languages and need built-in mitigation to avoid these vulnerabilities. With subsequent updates to Wasm specifications, WebAssembly can finally become the staple byte code of web development.