Cyberpeace Whitepaper

Is WebAssembly Really Safe? - Wasm VM Escape and RCE Vulnerabilities Have Been Found in New Way

July 19, 2022 – Version 1.0

Prepared by
Zhao Hai(@h1zhao)
Zhichen Wang
Mengchen Yu
Lei Li

Abstract

WebAssembly(Wasm) supports binary format which provides languages such as C/C++, C# and Rust with a compilation target on the web. It is a web standard with active participation from all major browser vendors (chrome, edge, Firefox, safari). Also, Wasm runtime can be widely used for edge computing.

Previous research on Wasm security mostly focuses on exploitation at the compiler and linker level, but few people focus on Wasm VM escape. Therefore, we design a new fuzz framework based on Wasm standard to explore the runtime vulnerability itself. The framework can be compatible with all programs or projects containing Wasm design standards.

If there is an escape vulnerability in the browser kernel or any project that uses Wasm runtime, when an attacker deploys a page or service containing a malicious Wasm binary, he can control the access device or the server that provides the runtime service.

We find that these escape vulnerabilities are usually caused by inadequate operand boundary checking of bytecode interpreter or stack overflow of WASI API. For example, in wasm3 and WasmEdge projects, we use the above two methods to achieve VM escape. Meanwhile, there are many exploitable vulnerabilities in the parsing of file data structure, which are usually overflow vulnerabilities caused by inadequate inspection of some input fields. Normally, these vulnerabilities will lead to denial of service attacks. In the process of fuzzing, we find that almost all wasm runtime projects can exploit such vulnerabilities

Finally, we will show the off-by-one vulnerability of a PC stack of WasmEdge that we discovered, which successfully conducts RCE on the host. This process is very ingenious and we will explain it in detail at the demo time.

天虞实验室

1 Introduction

WebAssembly is a technology developed by a W3C Community Group.The initial goal was to allow the developers to take their native C/C++ code to the browser, making the websites ran more faster.WebAssembly runs code in the virtual machine, so it is cross platform like Java.Now wasm is used in more fields, in the embedded field, it can be used to write platform independent programs,in the field of cloud computing, it can be used for program isolation, similar to docker. Therefore, it is significant to research WebAssembly Security.

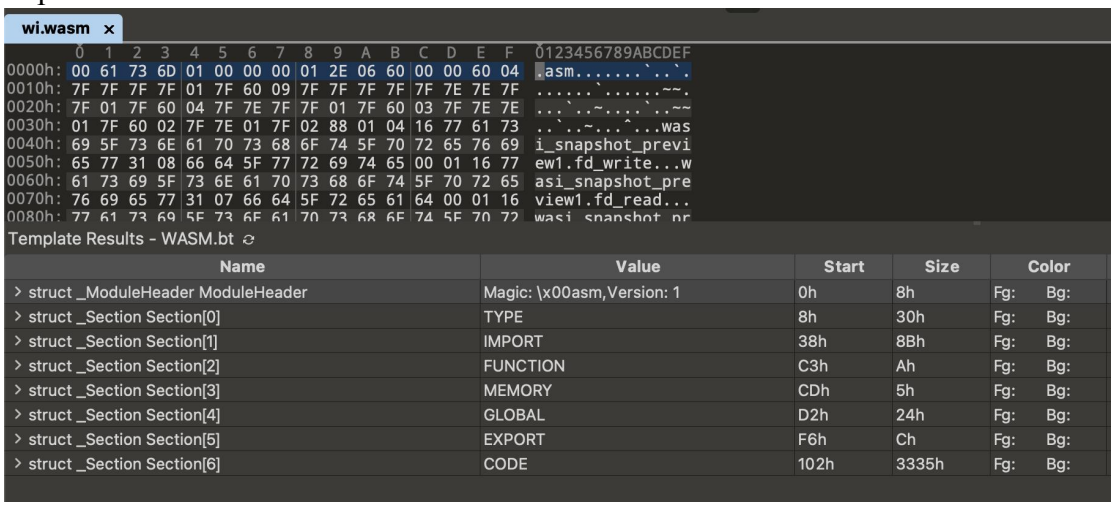
WebAssembly make the code runs safe, because it separate control flow stack and data stack. During execution, only the data stack can be modified, the control flow stack is fixed during parsing stage.So when a C/C++ program with a buffer overflow vulnerability compiled to wasm bytecode, it can't be exploitable. The summary is that all vulnerability overflows occur in the data stack and will not have any impact on the program flow stack.WebAssembly System Interface is a interface for WebAssembly to use System APIS, such as file operations, network.WASI is restricted by permissions, so it's safe.

Previous research on wasm security mostly focuses on exploitation at the compiler and linker level, but few people focus on wasm virtual machine escape.Our research focus on wasm virtual machine's vulnerability,we design a fuzzing tool to find vulnerabilities. Why we design a new fuzzing framework rather than existing fuzzing tool ? The wasm file has a certain data structure. When the wasm runtime validates, only the wasm samples that pass the validation can be executed in the next step. The samples generated by traditional tools such as AFL are raw binary files do not have contextual dependencies, which leads to no correlation between data structures, and the effect is not ideal.We analyse the wasm virtual machine's structure,there are three possible vulnerabilities in wasm virtual machine: one is the parser of wasm file, the second is the interpretation and execution of byte code of wasm, and the third is the WASI API. Our fuzzing tool is also designed based on these situations.

We found some exploitable vulnerabilities in wasm virtual machine. we will introduce the wasm virtual machine's architecture, we will show the vulnerability existed in wasi api and exploit it. we will show how to use bytecode to spray the heap, from a bytecode off by one to get code executions.

2.1 WASM Structure

Wasm consists of many sections, the CODE section stores wasm bytecode, the import section stores some imports' name string, the wasi is also imported from import section.



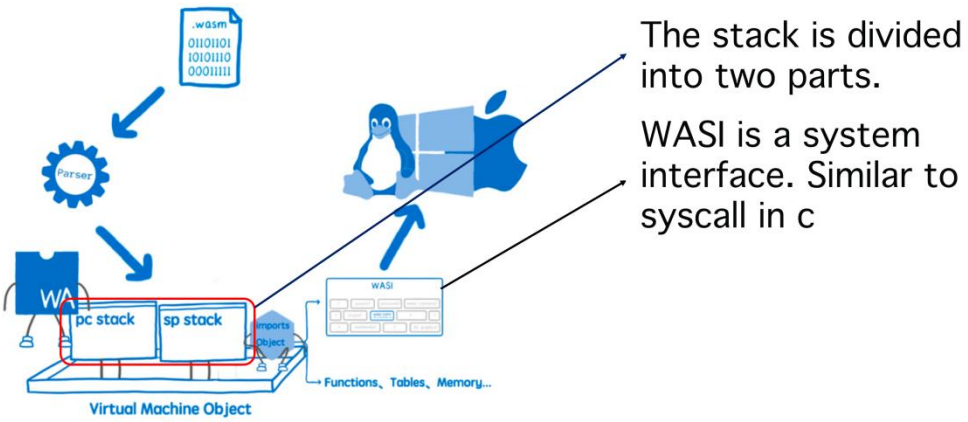
2.2 WASM S-Expression

The S-Expression can be understood by human beings,it's just like this

```
(module
(memory $0 1)
(export "memory" (memory $0))
(export "main" (func $main))
(func $main (; 1 ;) (result i32)
(i32.const 16191)
)
)
```

same as the wasm binary, it consists of many sections. The wasm bytecode is stack-based, for example, i32.const 16191 will push value 16191 to stack. i32.add will pop two values from stack to add and then push the result to stack.

2.3 WASM Virtual Machine



We focus on wasm file structure、wasi api、bytecode implemention in runtime.We develop a wasm sample generator.It constructs samples in sections,abstracts each section into a class structure and maintain it.

The design of generator adopts the idea of object-oriented, and splits the data structure of the entire WASM.

Wasm is composed of a variety of different sections, then design different classes to handle different sections, and each object is responsible for its own small part sample generation.

For the code section, it mainly stores some wasm bytecodes. Each bytecode is designed with a corresponding class, which is responsible for obtaining data from Random and generating some required operands. This class will also add some rules to limit the scope of the operand.

The operand of some bytecode may depend on some other objects in wasm, it's needed to design a Context class to store some context-related data for other objects to use, and finally output the binary data corresponding to this bytecode into the sample.

The quality of the sample generator directly affects the fuzz effect. In a data structure, if some places are constants, there is no need to detect new paths for mutation. For example, the API name of Wasi is some fixed strings, and mutation of these strings has no effect, so it is fixed; For example, if a field is a CRC check value, it cannot be generated correctly by random mutation, so you need to calculate the CRC of the data according to the rules and fill in the field later. The value of some fields is limited to a range, such as a string variable whose value is limited to a random one in the ["apple", "banana"] list, so the field only needs to be selected from the list when it is generated.

In the sample generator, we have implemented our own strategic random number generator.

Algorithm 1 random integer

```
Ensure: random integer
function INTEGER
    c ←range(0,6)
    switch c do
        case 0
            v ←0
            break
        ........
        case 6
            c2 ←range(0,7)
            switch c2 do
                ........
                case 4
                    v ←0x100000000
                    break
                case 5
                    v ←0xffffffff
                    break
                case 6
                    v ←0x80000000
                    break
                case 7
                    c2 ←range(0,50000)
                    break
            break
    return v
```

For example, in the algorithm in the figure above, according to previous experience, the most likely problem of integers is integer overflow, which often occurs near the boundary of integers. Therefore, it is necessary to make the probability of boundary values greater. Range in the figure is a random number generator, and which value to return is selected according to the value of range. Similarly, we encapsulate the generation algorithms of some basic data types, such as integer64, short, float32, float64, etc., and use strategic methods to make their boundary values appear more frequently.

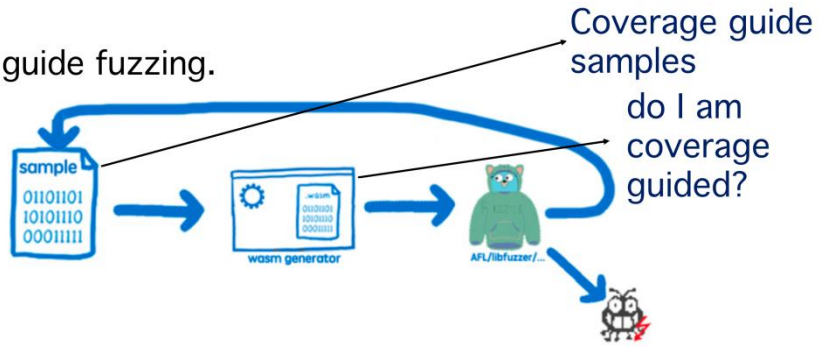
When the sample generator runs independently, the value of range comes from /dev/urandom, which is random. However, if you want to add it to AFL, libfuzzer and other tools and support coverage guidance, you only need to change the data source of range to read from the original binary data generated by AFL, libfuzzer and other tools. That is, all values are no longer autonomously random, but based on the samples of the fuzzer tool, which is equivalent to establishing a data mapping. Each data in the original data of the fuzzer tool uniquely corresponds to a state in the sample generator, which makes it possible for the sample generator to guide based on coverage.

Algorithm 2 Random

```
1:  function RAND_FROM_FUZZER
2:      x ←0
3:      if pos + 4 <= data_len then
4:          x ←getUint(data[pos])
5:          pos ←pos + 4
6:      return x
7:  function RANGE(LOW,HIGH)
8:      x ←rand_from_fuzzer()
9:      x ←(x % (high - low + 1)) + low
10:     return x
```

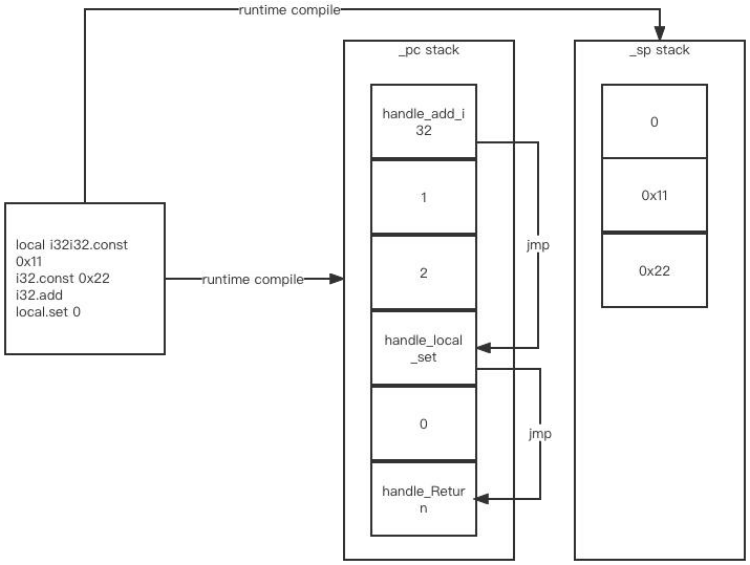
When the value of data is not enough, return the constant 0 instead of reading a data randomly from /dev/urandom, which is to ensure the uniqueness of the mapping. If the value of data is not enough to return a random number randomly, for the same data, the state it brings to the whole sample generator is not unique, and the coverage guidance cannot be completed.

- Focus on WASM file structure、WASI API、bytecode implemention in runtime.
- Coverage guide fuzzing.



4.1 WASM3

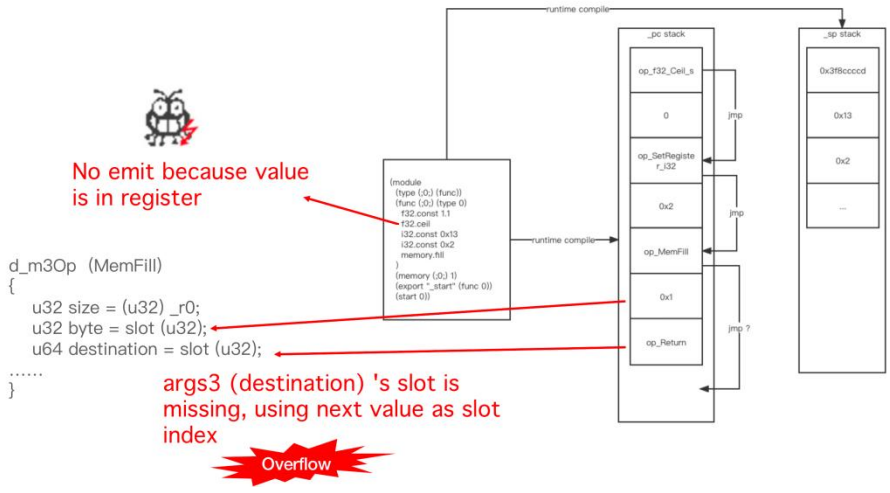
Wasm3 uses a _pc stack and _sp stack, where the pc stack stores a series of runtime functions and parameters corresponding to opcode the parameter in _pc stack uses slot index, which represents the parameter in the subscript in _sp, in the runtime function, from the value of the parameter was read in _sp.



Among them the data operations in wasm bytecode are in _sp stack, _sp stack only stores data, so stack overflow in the traditional sense, such as the operation similar to memcpy(buf, P, 0x10000) implemented by wasm bytecode, cannot affect the program flow of wasm bytecode.

Therefore, the design of wasm is safe, but if there is a problem with the wasm runtime itself, it will be unsafe.

For example, in the process of runtime compile, a handle needs 3 slots, but the compiler calculates the number of slots as 2 incorrectly, so the third slot value is the address of the next handle, and this value is used as a subscript to access _sp stack, which is a bytecode level vulnerability we found.

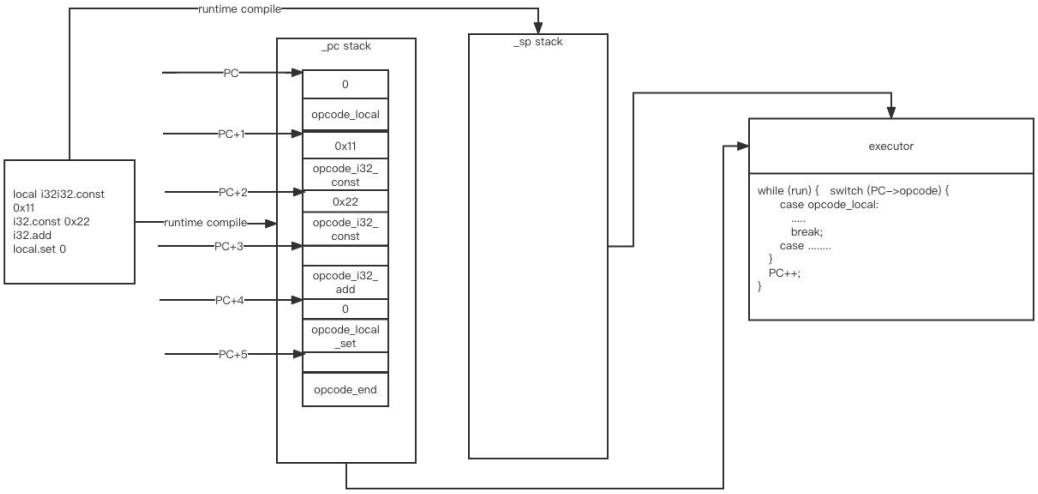


Wasm runtime is a virtual machine, so the idea of virtual machine escape can be used for reference. Virtual machines often escape through the vulnerabilities of

various devices. Under many wasm runtime, Wasi APIs are implemented. These APIs work on the host, so you can find such API vulnerabilities.

4.2 WasmEdge

The structural design of wasedge is also the separation of data stack and execution stack. In the execution stack, there are a series of instruction objects.



We discovered the off by one vulnerability of a PC stack in wasedge, and successfully conducted rce, from an off by one to a complete exp. this process is very clever, and we will explain it in detail at the demonstration meeting.

5 Conclusions

The wasm standard is designed to be secure, but the wasm runtime is not entirely secure.

The escape trigger of the Wasm VM is usually based on the bytecode interpreter and the WASI API, so we must focus on these two modules when designing. It's suggested to add more tests to check for bugs in these two aspects, strictly check the operands of opcode, and strictly check the parameters passed in by wasi api.

We will further study more wasm runtime projects in the future, and provide more support for related project security.