Vulnerabilities in WebAssembly: A Survey

Holger Klein
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany

Figure 1: Seattle Mariners at Spring Training, 2010.

ABSTRACT

A clear and well-documented LaTeX document is presented as an article formatted for publication by ACM in a conference proceedings or journal publication. Based on the “acmart” document class, this article presents and explains many of the common variations, as well as many of the formatting elements an author may use in the preparation of the documentation of their work.

KEYWORDS

binary exploits, Webassembly, IT Security

ACM Reference Format:
Holger Klein. 2021. Vulnerabilities in WebAssembly: A Survey. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

WebAssembly is a binary code format and compilation target meant to bring performance to web applications. The initial design of the API and binary format of WebAssembly got completed in 2017 [8]. Since then, most major browsers such as Firefox, Chrome or Safari implement many of the proposed features. Even in the backend it is possible to run code compiled to WebAssembly for example when using Nodejs. The promise of running code with near native performance in the browser is very attractive, as it allows for more demanding web applications and smoother user experiences. However, since any Website visited by the user can download and execute WebAssembly code just like Javascript, it immediately raises security concerns. On the one hand, a malicious website could execute malware on the host PC or use computing ressources by executing a crypto miner. On the other hand, a vulnerable WebAssembly program which takes user input could lead to cross site scripting attacks in the browser. Worse yet, as WebAssebly gets adopted in the backend or even in stand-alone applications, vulnerabilities in WebAssembly programs could enable attacks such as Remote Code Execution. This survey will mainly deal with issues of the latter kind, focusing on how WebAssembly programs might be vulnerable to binary exploits. In particular, we will focus on how the security mechanisms intrinsic to WebAssembly's design compare to binary exploit mitigations in native applications. The official spec adresses some of the security concerns by stating that "[...] code is validated and executes in a memory-safe*, sandboxed environment preventing data corruption or security breaches". However, it adds a footnote which specifies "*No program can break WebAssembly's memory model. Of course, it cannot guarantee that an unsafe language compiling to WebAssembly does not corrupt its own memory layout, e.g. inside WebAssembly's linear memory". Indeed, in the past years there have been a few puplications commenting on WebAssembly's lack of mitigations to common binary exploitation techniques, such as [5] or [4]. Additionally, there have been puplications researching the real-world assembly programs such as [6] or [3]. In this survey, we aim to cover the main points of these publications and try to recreate their main results. It is structured as follows: 2 will introduce WebAssembly with a focus on the features which will be important to our discussion later. ASDASDASDASD

2 WEBASSEMBLY

The following will give an Introduction to and and overview of WebAssembly, paying special attention to the parts relevant to a discussion of binary vulnerabilities. For more information, see the official specification at [7], or the Background section in [4].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference'17, July 2017, Washington, DC, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2.1 High Level Overview

The name 'WebAssembly' (often abbreviated as WASM) is a slight misnomer, since it has a different form and function from typical assembly languages. The official spec refers to it as "low-level, assembly-like". It is a binary byte code format which is interpreted by a Virtual Machine, similar to for example Java. The Virtual Machine is most often implemented in a browser. Design goals were to make WebAssembly safe, hardware- and language independent and fast. In fact, WebAssembly is supposed to run at near-native speeds. There exists a human-readable format of WebAssembly binaries called 'wat'. Whenever we present WebAssembly Code, it will be in this format. While it is possible to hand-craft wasm binary, it is most often generated by compiling a high-level language such as C/C++ or Rust. The Binary is then instantiated by calling a Javascript function. See **??**. By itself, a compiled WebAssembly module has now way of communicating with the host environment such as a Browser (assuming a Bug-free Virtual Machine). WebAssembly functionality can only be accessed by calling functions which are exported by the WebAssembly module. These exported functions can be called from Javascript code. Conversely, WebAssembly has no I/O other than what is directly supplied through imported Javascript functions.

```
WebAssembly.instantiateStreaming(fetch('myModule.wasm'), importObject)
.then(obj => {
  // Call an exported function:
  obj.instance.exports.exported_func();

  // or access the buffer contents of an exported memory:
  var i32 = new Uint32Array(obj.instance.exports.memory.buffer);

  // or access the elements of an exported table:
  var table = obj.instance.exports.table;
  console.log(table.get(0)());
})
```

Figure 2: How to intantiate a WebAssembly module using Javascript. (https://developer.mozilla.org/en-US/docs/WebAssembly/Loading_and_running).

2.2 Modules

At the highest level, WebAssembly programs are organized into Modules. A module is what gets compiled and run by the Virtual Machine. Amongst other things, a module contains definitions for imports and exports, functions, types, tables and memories. All definitions are referenced by zero-based indices. As of the time of writing, only one memory may be defined in a module. This memory is indexed by 0 and implicitely referenced by all other constructs. More on memory in 2.5. Also as of time of writing the only table elements which are available are untyped function references. This is used to implement indirect function calls, see 2.6. For an example of WebAssembly code, see

2.3 The Stack

The WebAssembly virtual machine is Stack based. Thus, it doesn't have registers. Instead, values are pushed on and popped of the stack. All instructions implicitely operate on the stack.

```
char * str = "Hello world\n";

char * indirect_func() {
  return str;
}

WASM_EXPORT
char *direct_func(int i) {
  char * (* ptr)(void)  = &indirect_func;
  return (*ptr)();
}
```

Figure 3: A c program, which, when compiled without optimization, gets translated to the code in Figure 4.

```
(module
  (type $t0 (func (result i32)))
  (type $t1 (func))
  (type $t2 (func (param i32) (result i32)))
  (func $__wasm_call_ctors (type $t1))
  (func $indirect_func (type $t0) (result i32)
    (local $l0 i32) (local $l1 i32)
    [...]
    i32.load offset=1040
    [...]
    return)
  (func $direct_func (export "direct_func") (type $t2) (param $p0 i32) (result i32)
    [...]
    call_indirect (type $t0)
    [...]
    return)
  (table $T0 2 2 anyfunc)
  (memory $memory (export "memory") 2)
  [...]
  (elem (i32.const 1) $indirect_func)
  (data (i32.const 1024) "Hello world\0a\00")
  [...]
```

Figure 4: The c program in Figure 3 gets compiled to this WebAssembly program (with uninteresting parts removed). Observe the function table and memory.

2.4 Control Flow

In contrast to native languages or even Java, WebAssembly enforces structured control flow. Code can only be organized into blocks. Control-flow commands can only jump to the beginning of such a block, and only within the current function. In addition, the bytecode never interacts with the underlying adresses of functions. These are only accesible to the Virtual Machine. This immediately makes many binary exploits infeasable, such as Return-Oriented Programming.

2.5 Memories

WebAssembly only supports four different primitive types: 32- and 64-bit Integers (i32, i64) and single- and double-precision floating point nubmers (f32, f64). As such, arrays, pointers, etc. have no explicit representation at the binary level. To see how they are implemented in binary programs, it is necessary to understand the way WebAssembly handles memory. The implementation of function

pointers is discussed in 2.6. The Stack of a function which contains the return adress as well as any native data type whose adress is never taken is not accesible to the bytecode. As mentioned in 2.4, this in itself contributes to WebAssemblys inherent security. However, anything other than a value of native type or any value whose adress is taken needs to reside in the linear memory. This memory is a linear array of bytes. Adresses are referenced by 32 bit integers which serve as pointer types. The WebAssembly program can request more memory from the Host VM using the memory.grow operation. The linear memory is completely unmanaged. The way it is used is completely up to the program. Many WebAssembly toolchains such as Emscripten include an allocator which implements functions such as C's malloc and free. This unmanaged, linear memory is the main vector of attack of all vulnerabilities discussed in [4]. As will be discussed in section 3.1.

2.6 Indirect Function Calls

To implement indirect function calls, any function which may get called indirectly or used as a function pointer has an entry in the table section. The **call_indirect** operation pops an index from the stack which is used to reference an entry in the table section. This table maps the index to a function. This limits which functions can be called indirectly. Additionally, functions in WebAssembly are type checked. The **call_indirect** operation has the function type statically encoded. Thus, an indirect call can only call functions which have the same signature. It must be remarked however that this is less limiting than it might first appear, since WebAssembly only has four native types. Thus, a function taking a pointer (or a string) has the same signature as a function taking an 32-bit Integer.

2.7 Deployment, Compilers and Toolchains

It would be very impractical (albeit possible) to write WebAssembly from scratch. Thus, several Backends exist to generate WebAssembly bytecode from high-level languages such as C, Go or Rust. Emscripten [1] can not only generate the Bytecode from C/C++ but also html and Javascript to load and run the WebAssembly module. It also comes with C Headers to interact with the Browser and implements several common libraries such as SDL2 [2].

3 BINARY VULNERABILITIES OF WEBASSEMBLY PROGRAMS

In this section we will discuss the security vulnerabilities of WebAssembly as presented in [5] and [4]. [4] begin by discussing the security related aspects of the linear unmanaged memory. As mentioned in 2.5, every scalar value whose adress is never taken, as well as function return adresses are completely controlled by the virtual machine. This mitigates many well-known attacks. However, all non-scalar types such as arrays or any value whose adress is ever taken must lie in the unmanaged memory. This memory is usually controlled by an allocator provided by the compilation toolchain. Most allocators seperate the memory in three distinct regions: The Stack, Heap and Data sections. To distinguish between the function call stack managed by the VM and the call stack created by the compiler, [4] call the latter the **unmanaged** stack. We will use the same nomenclature here. One must keep in mind however, that there are no underlying mechanisms provided by the VM to seperate between these three regions. This seperation is only implemented by the compiler. Initially, it is just a single contiguous linear array of bytes.

3.1 Exploit potential of unmanaged memory

[5] discuss serveral common exploit mitigation techniques which are used in binary programs.

Exploit mitigation	Protect against
Address Space Layout Randomization (ASLR)	Attacks on Control-Flow
Stack Canaries	Stack-based Buffer overflo
Heap Hardening	Manipulating Allocator Meta
Data Execution Prevention (DEP)	Code injection

REFERENCES

[1] [n.d.]. . https://emscripten.org/
[2] [n.d.]. . https://www.libsdl.org/index.php
[3] Aaron Hilbig, Daniel Lehmann, and Michael Pradel. 2021. An Empirical Study of Real-World WebAssembly Binaries. (2021), 13.
[4] Daniel Lehmann, Johannes Kinder, and Michael Pradel. 2020. Everything Old is New Again: Binary Security of WebAssembly. *Proceedings of the 29th USENIX Security Symposium* (Aug. 2020), 217–234.
[5] Brian McFadden, Tyler Lukasiewicz, Jeff Dileo, and Justin Engler. 2018. Security Chasms of WASM.
[6] Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. 2019. New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild. (2019), 20.
[7] Andreas Rossberg. 2021. *WebAssembly Specification.* https://webassembly.github.io/spec/core/
[8] Nick Schonning. 2021. *Loading and running WebAssembly code.* https://webassembly.org/roadmap/