Vulnerabilities in WebAssembly

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

ABSTRACT

WebAssembly is a binary bytecode format and compilation target originally designed for use together with Javascript to bring almost native performance to the web. Security mechanisms inherent to the language were a major focus during the design. Structured control flow and managed function return addresses make many popular exploits such as return-oriented programming impossible. However, recent publications claim to show that due to the absence of common exploit mitigation techniques, vulnerabilities in high-level languages such as C lead to vulnerabilities in compiled WebAssembly. We introduce the main vulnerabilities and demonstrate proof-of-concept exploits. An overview over the most promising software analysis techniques such as abstract interpretation or fuzzing is presented. We introduce the most up-to-date analysis on real-world WebAssembly binaries. It seems to suggest that many deployed programs have some of the discussed vulnerabilities. Despite this, there are no known widely circulated exploits. We present some reasons as to what might explain this discrepancy.

KEYWORDS

binary exploits, Webassembly, IT Security

1 INTRODUCTION

Introduced in 2018, WebAssembly is a still novel binary code format and compilation target originally meant to bring performance to web applications. The initial design of the API and binary format of WebAssembly got completed in 2017 https://webassembly.org/roadmap/. Since then, most major browsers such as Firefox, Chrome or Safari implement many of the proposed features. WebAssembly has since also been implemented on other platforms, even on the server it is possible to run code compiled to WebAssembly for example 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 computer or use computing resources 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 work will mainly discuss the latter, focusing on how WebAssembly programs might be vulnerable to binary exploitation techniques. In particular, we will focus on how the security mechanisms intrinsic to WebAssembly's design compare to binary exploit mitigations in native applications. As such, this can be seen as a comparison between exploits in native binaries and WebAssembly binaries.

Security is a major selling point of WebAssembly as a platform and was a big concern when it was designed. One publication notes "At worst, a buggy or malicious WebAssembly program can make a mess of the data in its own memory" [12]. The official spec itself addresses 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 publications commenting on WebAssembly's lack of mitigations for common binary exploitation techniques, such as McFadden et al. [10] or Lehmann et al. [5]. Additionally, there have been puplications researching the real-world assembly programs such as Musch et al. [11] or Hilbig et al. [4]. In this work, we aim to cover the main points of these publications and comment on how well WebAssembly is protected against binary exploits by design. We close by presenting the to date biggest analysis on real-world binaries and discuss the apparent lack of widely circulated exploits.

2 BACKGROUND

This survey aims at researching the binary vulnerabilities of WebAssembly. To this end, it is imperative to compare how security mechanisms inherent to WebAssembly compare to security concerns of native binaries. There exists a host of different binary exploitation techniques, and securing against them seems an ever changing arms race between security professionals and hackers. We will mainly concern ourselves with vulnerabilities which are exploited through some malicious input provided by an attacker. These inputs can be a string, file, or key sequence which somehow triggers unexpected behavior in our program. To do so, we will first give an introduction to WebAssembly and how it, as a compilation target, differs from native binary formats and execution environments. Then, we will discuss how this influences typical vulnerabilities such as Stack or Buffer overflows.

2.1 WebAssembly

The following will give an Introduction to 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 https://webassembly.github.io/spec/core/, or the Background section in Lehmann et al. [5].

Vuln.Disc.Colloquium '21, July 23, 2021, KIT, Karlsruhe, Germany

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 (VM), similar to for example Java. The VM 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 a wasm binary, it is most often generated by compiling a high-level language such as C/C++ or Rust. In the browser, the binary is then instantiated by calling a Javascript function (see Listing 1). By itself, a compiled WebAssembly module has now way of communicating with the host environment such as a Browser (assuming a Bug-free VM). Functionality implemented in a WebAssembly module can only be accessed by calling functions which are exported by the 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)());
})
```

Listing 1: How to instantiate 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 the unit which gets compiled and run by the VM. Among 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 implicitly referenced by all other constructs. Memory will be discussed in Section 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 Section 2.6. For an example of WebAssembly code, see Listing 3, which the code in Listing 2 get compiled to.

2.3 The Stack

The WebAssembly VM is conceptually stack based. In contrast to most native platforms, it doesn't have a notion of registers. Instead, values are pushed on and popped off the stack. Many instructions implicitly operate on the stack, such as `call_indirect` which calls a function indirectly based on the index pushed onto the stack (See Listing 3).

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

char *indirect_func() {
  return str;
}

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

Listing 2: A C program which, when compiled without optimizations, gets translated to the code in Listing 3.

```
(module //start of the module, the unit of compilation in WebAssembly
  (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) //a function definition
    (local $l0 i32) (local $l1 i32)
    [...]
    i32.load offset=1040

    [...]
    return9
  (func $direct_func (export "direct_func") (type $t2) (param $p0 i32) (result i32)
  (local $l0 i32) (local $l1 i32) (local $l2 i32) (local $l3 i32) (local $l4 i32) (local $l5 i32) (local $l6 i32) (local $l7 i32)
  [...]
  call_indirect (type $t0) //calling a function indirectly based on the index pushed onto the stack
  [...]
  return)
(table $T0 2 2 anyfunc)
(memory $memory (export "memory") 2)
(elem (i32.const 1) $indirect_func) //indirect_func can get called indirectly by pushing '1' onto the stack
(data (i32.const 1024) "Hello world\0a\00") //A string in memory. There exists no constant storage in WebAssembly.
(data (i32.const 1040) "\00\04\00\00"))
```

Listing 3: An example of a WebAssembly program in the human-readable `wat` format. The C program in Listing 2 gets compiled to this WebAssembly program (with uninteresting parts removed). Observe the indirect call, function table and memory region.

2.4 Control Flow

In contrast to native languages or even Java, WebAssembly enforces so-called 'structured' control flow. Code is organized into blocks. Control-flow commands such as those generated by loops or conditions can only jump to the beginning of such a block, and only within the current function. In addition, the bytecode has no way of interacting with the underlying addresses of functions. These are only accessible to the VM. This immediately makes many binary exploits impossible, 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 numbers (`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 or virtual functions is discussed in Section 2.6. The stack space of a function which contains the caller's return address as well as any native data types whose address is never taken cannot be accessed by the program. As mentioned in Section 2.4, this in itself contributes to WebAssemblys inherent security. However, anything other than a scalar value or any value whose address is taken needs to reside in the so-called linear memory. This memory is a linear array of raw bytes. Addresses are referenced by 32-bit integers which serve as the pointer type. The WebAssembly program can request more memory from the Host VM using the `memory.grow` operation. The linear memory is completely unmanaged by the VM. The way it is used is completely up to the program. Many WebAssembly toolchains such as emscripten (https://emscripten.org/) include an allocator which implements functions such as C's `malloc` and `free`. This unmanaged, linear memory is the main reason for most vulnerabilities discussed by Lehmann et al. [5]. This will be elaborated on in Section 3.

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 module's table section. The `call_indirect` operation pops an index from the stack which is used to reference an entry in the function table section. This table maps the index to a function. This limits which functions can be called indirectly, as the only functions available are the ones in this table. Listing 4 shows example code to make this more intuitive. The code prints `1,2,1,3` to the console when running it in the browser, as those are the functions' indices in the function table.

```c
#include <stdio.h>
#include <emscripten.h>

void printLine(const char *s) {
  printf("%s\n", s);
}

void printInteger(const int i) {
  printf("%i\n", i);
}

int main() {
  printf("%i\n", &printLine);
  printf("%i\n", &printInteger);
  printf("%i\n", &printLine);
  printf("%i\n", &emscripten_run_script);
  return 0;
}
```

Listing 4: This Code demonstrates how WebAssembly handles function pointers, by referring to functions using their assigned indices in the module's function table. This code prints `1,2,1,3` to the console when running it in the browser.

Additionally, functions in WebAssembly are type checked. The `call_indirect` operation has the function type statically encoded. Thus, this operation can only call functions which have the matching 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. This is demonstrated by the code in Listing 5. By copying the index of different functions directly into the function pointer, different functions get called indirectly. The function referenced by the function pointer expects as argument a pointer to a `char`. However, it is possible to use the index of a function which expects an integer.

```c
#include <stdio.h>
#include <emscripten.h>
#include <cstring>

struct FunctionStruct {
  void (*f) (const char *);
};

void printLine(const char *s) {
  printf("%s\n", s);
}

void printInteger(const int i) {
  printf("%i\n", i);
}

int main() {
  //using the functions' adresses so they get put in
   the function table
  printf("%i", &printLine);
  printf("%i", &printInteger);
  printf("%i\n", &emscripten_run_script);

  FunctionStruct fs;
  char *printLineIndex = "\x01\x00\x00\x00";
  memcpy(&fs, printLineIndex, 4);
  //prints "printing line" to the console
  fs.f("printing line");

  char *printIntegerIndex = "\x02\x00\x00\x00";
  memcpy(&fs, printIntegerIndex, 4);
  //prints 1059 to the console
  fs.f("");

  char *runscriptIndex = "\x03\x00\x00\x00";
  memcpy(&fs, runscriptIndex, 4);
  //displays an alert
  fs.f("alert('alert')");

  return 0;
}
```

Listing 5: Example demonstrating how function pointer type checking works. Even though `FunctionStruct` has function `f` as a member which expects a pointer to a `char`, it can be overwritten with the index of a function which expects a 32-bit integer. This is due to both pointers and 32-bit integers being represented by the same datatype in WebAssembly, namely `i32`

2.7 Deployment, Compilers and Toolchains

It would be very impractical (albeit possible) to write WebAssembly from scratch. Thus, several tools exist to generate WebAssembly bytecode from high-level languages such as C, Go or Rust. emscripten can not only generate WebAssembly bytecode from C/C++

but also generate 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 SDL1 https://www.libsdl.org/ and a wrapper to simulate Linux sockets using WebSockets.

3 VULNERABILITIES OF UNMANAGED MEMORY

In this section we will discuss the main security vulnerability of WebAssembly as presented by McFadden et al. [10] and Lehmann et al. [5]. Lehmann et al. [5] begin by discussing the security-critical aspects of the linear unmanaged memory. As mentioned in Section 2.5, every scalar value whose address is never taken, as well as function return addresses are completely managed by the VM. This mitigates many well-known attacks. However, all non-scalar types such as arrays or any value whose address is ever taken must lie in the unmanaged memory. This memory is usually controlled by an allocator provided by the compilation toolchain. Most allocators separate the memory in three distinct regions: The stack, heap and data sections, similar to binary formats such as ELF. One must keep in mind however, that there are no underlying mechanisms provided by the VM to differentiate between these three regions. This separation is only implemented by the allocator. From the point of view of the platform, it is just a single contiguous linear array of bytes. To distinguish between the function call stack managed by the VM and the call stack created by the allocator, Lehmann et al. [5] call the latter the **unmanaged** stack. We will use the same nomenclature here.

3.1 Exploit mitigations in native and Webassembly binaries

McFadden et al. [10] discuss several common exploit mitigation techniques which are used in binary programs. Table 1 provides a summary and shows whether these techniques or some replacement are present in WebAssembly.

Lehmann et al. [5] also discuss common mitigations such as ASLR and page protection flags which are not present in WebAssembly. In particular, since there are no guard pages between the different sections of the linear memory, an overflow in any section can corrupt data in any other section. And since there is no concept of read-only memory, even data which is marked as constant in the source code can be overwritten during execution. In summary, the unmanaged, linear memory has the biggest potential to lead to exploits.

4 EXPLOIT POTENTIAL OF UNMANAGED MEMORY

Having analyzed WebAssemblys main vulnerability, the unmanaged linear memory, there are several ways to exploit it. McFadden et al. [10] show two attack primitives, format-string attacks and stack-based buffer overflows. They use these attack primitives to implement a proof of concept of a cross-site scripting attack, and remote code execution on a server. Lehmann et al. [5] similarly start by demonstrating two write primitves: First, they introduce a stack-based buffer overflow. Secondly, they also demonstrate how the allocator supplied by emscripten (called `emmalloc`) is susceptible to the so-called 'unlink exploit'. Given the right circumstances, this can be utilized to allow an attacker to write an arbitrary value to an arbitrary address. However, their exploit didn't work on our machine using version 1.39.16.of emscripten. They also describe how malicious, user-supplied input data could lead to a classical call-stack overflow. For example, a program which expects as input an acyclic graph could be supplied with a cyclic graph. Since WebAssembly has no guard pages, such a stack overflow could overwrite data following the stack such as the heap or even read-only data. This depends on the memory layout introduced by the allocator. However, no example of an exploit is given.

We will quickly summarize the presented attack primitives. One of the main sources of security of WebAssembly comes from the fact that it is isolated from the surrounding environment by the VM. Thus, for an exploit to be dangerous in the browser, it needs to manipulate data that gets passed to functions which can have an effect outside the isolated program. In the browser, such functions will be imported from Javscript. Examples are functions such as `eval` or functions which manipulate the DOM. The Javascript function `eval` will execute any string passed to it as Javascript code.

4.1 Overflowing the unmanaged stack

Overflowing buffers on the stack is one of the most widely used binary exploitation techniques. It can happen whenever user-supplied data is copied into a buffer on the stack (such as a fixed-size array) without checking the bounds. The most common example of a function which has this vulnerability is the C function `gets`. The Linux manual even states, plainly, "never use `gets`". There are many ways to exploit a stack based buffer overflow. In native binaries without exploit mitigation techniques such as stack canaries, a buffer overflow can be used to overwrite the functions return address. This can allow an attacker to execute arbitrary code with the permissions of the program. However, since the WebAssembly VM manages return addresses separately, one could assume that stack based buffer overflows aren't security concerns for WebAssembly programs. However, if the function has non-scalar data, this will reside in unmanaged memory. In fact, since there exist no stack canaries in WebAssembly, a buffer overflow can overwrite any unmanaged memory above the function's stack frame. This can even include parent frames. Additionally, there are no guard pages in WebAssembly so a stack based buffer overflow can grow into other memory regions such as the heap. And, since there is no concept of read-only data in WebAssembly, any data on the unmanaged stack can potentially be overwritten. Lehmann et al. [5] have several examples to demonstrate this vulnerability. Listing 6 shows an example inspired by their work. The `main` function generates a string which gets added to the DOM. Before that, it calls a vulnerable function. The user provided input can overflow the stack of the vulnerable function and into the stack of the `main` function. There, it can overwrite the string added to the DOM. This can lead to a cross site scripting attack.

Exploit mitigation	Protect against	Effect in Native Binaries	WebAssembly
Address Space Layout Randomization (ASLR)	Attacks on Control-Flow	Randomizes the base address of an executable as well as the heap and stack addresses and addresses of libraries. This is meant to make exploitation techniques such as return oriented programming harder. It also makes exploiting other vulnerabilities more challenging	ASLR has no equivalent mechanism in WebAssembly. However, since the user can't access return addresses, many exploits that rely on changing control flow are impossible. In addition, since WebAssembly only provides 32-bit addresses, it is assumed to have too little entropy for ASLR to be effective. Also, since functions are directly accessed by indices instead of memory addresses, ASLR could not be used to obfuscate function addresses.
Stack Canaries	Stack-based Buffer overflows	Stack Canaries are placed on the stack such that a stack-based buffer overflow will overwrite them before corrupting the functions return address. This enables the program to detect and handle these overflows	Stack canaries don't exist in WebAssembly, since the return addresses are entirely managed by the VM
Heap Hardening	Manipulating Allocator Metadata	Heap Hardening comprises several different programming techniques to make allocators more secure against attacks. By corrupting heap metadata, an attacker can use the allocator to execute arbitrary writes on the heap.	Since size is a concern, many toolchains come with the options to compile with their own, smaller allocator which might have vulnerabilities.
Data Execution Prevention (DEP)	Code injection	This mitigation is part of a family of mitigation techniques which modify memory pages to only allow certain behavior. These can, for example, modify whether the contents of certain memory pages can be executed. In native binaries, this can prevent the injection of malicious code.	In WebAssembly, there is a strict separation between code and data. Hence, DEP is not needed (or in other words, always implemented).

Table 1: A summary of exploit mitigation techniques commonly found in native binaries. They are compared to their WebAssembly counterparts, if any exist.

```c
#include <emscripten.h>
#include <stdio.h>
#include <string.h>

void vulnerable() {
  char buffer[8];
  //imagine getting this from the user
  char *data = "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa<script>alert('hello')</script>";
  strcpy(buffer, data);
}

void generateHTML(char *buffer) {
  char *welcomeMsg = "<p>welcome to my website</p>";
  strcpy(buffer, welcomeMsg);
}

int main() {
  char html[512];
  generateHTML(html);
  vulnerable();
  add_to_dom(html);
  return 0;
}
```

Listing 6: This C program has a stack overflow vulnerability. This can be exploited by the user supplying a buffer which overflows into the parent frame and replaces the string meant to be added to the DOM.

4.2 Redirecting Control Flow

Can we influence the program control flow using an overflow as introduced in section 4.1? Since the return addresses are managed by the VM, the answer seems to be no. However, it is possible to overwrite indirect function calls. As discussed in Section 2.6, WebAssembly references functions by a 32-bit integer. The indirect function call can be influenced if two conditions are met:

- The integer which identifies the called function lies somewhere in unmanaged memory.
- The originally called function has the same signature as the new function we want to call instead.

Also, if we wish to influence the parameters passed to the newly called function they too must be within unmanaged memory. The most interesting fact is that, since WebAssembly has no way to mark memory as read-only, it is possible to overwrite supposedly constant data. McFadden et al. [10] demonstrate a proof of concept of using an overflow to manipulate control flow. We couldn't recreate their exact example, however a slight modification did work. The example in Listing 7 works on Firefox version 88.0.1 64-Bit when compiled with emscripten version 1.39.16.

The proposed scenario is that of a legacy application being ported to WebAssembly. It is possible to send messages and handle them using different functions. An especially crafted payload can be

used to overflow into the `msg_len` and `out` fields, changing the program behavior. It exploits an unchecked memory copy to alter the control flow. This is used to implement a cross-site scripting attack. When the code is executed in the browser, it displays an alert. Of course, one can imagine reworking this exploit to steal cookies, etc. The example is particularly interesting since it could only work in WebAssembly and not on other native platforms.

```
#include <stdio.h>
#include <emscripten.h>
#include <stdint.h>
#include <cstring>

struct Communication {
  char msg[64];
  uint16_t msg_len;
  void (*out) (const char *);
};

void printCommunication(Communication * comms) {
  comms->out(comms->msg);
}

void printLine(const char *msg) {
  printf("%s\n", msg);
}

int main() {
  //use adress hereso it gets added to the import table
  printf("%i\n", &emscripten_run_script);

  Communication comms;
  comms.out = &printLine;

  char *payload = "alert('XSS');// "  //16 byte attack
   script
          "                "  //16 byte padding
          "                "  //16 byte padding
          "                "  //16 byte padding
          "  "          //2 byte padding
          "\x40\x00"           //msg_len=68
          "\x01\x00\x00\x00"; //out=1

  memcpy(comms.msg, payload, 72);
  printCommunication(&comms); //trigger the exploit

  return 0;
}
```

Listing 7: This code demonstrates a proof of concept of an exploit using indirect calls to influence control flow and implement cross site scripting. The payload overrides the `msg_len` and `out` fields, redirecting control flow to implement a cross-site-scripting attack. This exploit could only work on WebAssembly and not on native binaries. Taken with modifications from McFadden et al. [10].

5 BINARY ANALYSIS

5.1 Analysis of native Binaries

Trying to decide whether a given program will crash is equivalent to the Halting problem. It follows that automating the discovery of binary vulnerabilities is a hard problem. We will first give some background on different techniques that have been developed for native binaries. Then, we will show which solutions exist for WebAssembly. A thorough summary of these techniques is outside the scope of this work, however we aim to give an intuitive understanding for the techniques used in the state of the art. See for example Tan et al. [15] for a more thorough review.

Techniques to discover binary vulnerabilities can roughly be divided into two categories: Static and dynamic analysis techniques [13]. A concept which will be interesting in this discussion is representing programs as Control Flow Graphs (CFGs). In such a graph, the nodes are instructions and the edges possible execution flows between these instructions. There exist algorithms to automatically generate CFGs from binary programs. This is, however, not an exact science. One problem is deciding how to trade-off false positive edges. This is especially the case when dealing with indirect or conditional jumps. While a CFG with zero edges would have no false positives, a CFG with edges between any two program instructions would surely contain all possible jumps as well as many false positives. Clearly, the ideal is somewhere in between these two extremes.

```
void f() {
  while(n < 1000) {
    n := n + 1;
  }
}
```

Listing 8: This pseudocode corresponds to the control flow graph in Figure 1

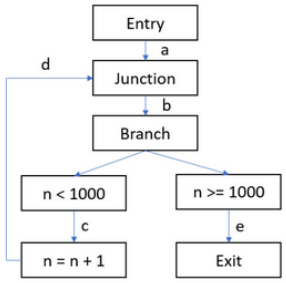

Figure 1: A Control Flow Graph derived from the Pseudocode in Listing 8

Static program analysis techniques reason about the program without executing it. Thus, the analysis has to reason about the properties of the program by abstracting it. The biggest analysis trying to find vulnerabilities in real-world WebAssembly programs undertaken thus far utilizes static analysis with a heuristic. It will be discussed in Section 6.

Dynamic binary analysis techniques on the other hand work by executing the program for some input and observing its behavior. Generating these inputs is most commonly done by a tool called a Fuzzer [7]. Fuzzers aim to generate inputs that trigger interesting behavior in the target program. Creating a Fuzzer which generates input that successfully triggers crashes is far from trivial. At a high level, Fuzzers are either coverage or taint based. Coverage based Fuzzers aim at generating inputs which maximize the percentage of executed code. Taint based Fuzzers analyze which parts of the code are influenced (tainted) by user input. These analyses require some understanding of the source program. It seems natural that effective

Fuzzers utilize some kind of static analysis. Thus, the separation into static and dynamic techniques isn't very strict.

An analysis method which lies somewhere in between static in dynamic methods is is abstract interpretation. Abstract interpretation is a based on programming language semantics and their fixed points [2]. Intuitively, the program is run for all values of a given type simultaneously. For example, instead of running a program using integers as a type, intervals are used. The operands defined on the original type are also adjusted to act on the abstract value type.

A popluar open-source framework to implement binary analysis techniques is the python-based `angr`, introduced by Shoshitaishvili et al. [13]. It provides tools for loading binaries, computing CFGs, etc. Boudjema et al. [1] build on `angr` to implement `VYPER`, a tool which utilizes concolic execution of the binary target. This is done to lower the amount of false positives.

There has also been work trying to utilize deep neural nets to locate faults in native binaries, for example by Li et al. [8].

5.2 Analysis of WebAssembly Binaries

There are several tools and approaches to specifically analyze WebAssembly Binaries to locate vulnerabilities. One proposed tool, Wasmati, is introduced by Lopes [9]. It aims to utilize Code Property Graphs. While promising, it has yet to be implemented and tested. TaintAssembly, introduced by Fu et al. [3] tracks taint through binaries using a modified V8 Javascript Engine. This comes with some limitations however such as not propagating indirect taint. It also seems like it wasn't ever evaluated on finding vulnerabilities in real-world vulnerabilities. Sun et al. [14] describe a deep neural net architecture to find vulnerabilities in WebAssembly binaries, the WASP framework. However, it has yet to be evaluated on more real-world binaries. Preliminary results seem promising, however.

Wasabi [6] is a Rust-based tool aimed at enabling the writing analyses of WebAssembly binaries. The analysis code is written in Javascript by utilizing hooks provided by the framework. The WebAssembly instructions are grouped to make writing analyses easier. For example, there is one hook for all function calls. If present, a user-implemented callback function is triggered on WebAssembly function calls while the function name and other information gets passed to the callback. Only those hooks which are implemented in the Javscript analysis get put into the binary to improve performance. Wasabi is the basis for the static analysis tools presented by Lehmann et al. [5] and also used by Hilbig et al. [4]. This tool performs static analysis utilizing heuristics to try and find possible vulnerabilities. It works as follows: Using the Rust-based Wasabi implementation as a library, the stack pointer in a given binary is identified. This is done using a heuristic which looks for a variable of type `i32` which is mutable, written to and read from globally, and has at least three reads and writes to avoid false positives in small binaries. This heuristic seems to work well when checked against manual verification. To identify possible vulnerabilities the tool looks for either of the following: use of unmanaged memory as introduced in Section 2.3, nonstandard and potentially unsafe memory allocators such as `emmalloc` (see Section 3) or importing potentially dangerous functions such as `eval`.

6 ANALYSIS OF REAL-WORLD WEBASSEMBLY BINARIES

To date, there exist three major analyses of different aspects of WebAssembly binaries "in the wild": One from 2019 [11], focused on the way WebAssembly is used by the 1 million most visited websites. Lehmann et al. [5] use the WebAssembly static analysis tool explained in Section 5.2 on real-world WebAssembly programs to ascertain whether the vulnerabilities they discuss exist in deployed code. Finally, in 2021 Hilbig et al. [4] ran the thus far largest analysis combining the methods of both Musch et al. [11] and Lehmann et al. [5]. Since their results refine the previous work and comprise a much larger dataset (8,461 unique WebAssembly binaries), we will focus mainly on their work. Afterwards, we will discuss their findings in Section 6.1.

Regarding the use of the unsafe, unmanaged memory, 65% of the analyzed programs make use of it (see Figure 2). Looking at the use of potentially unsafe memory allocators, only 11 of the analyzed binaries use `emmalloc`. 38,6% of all binaries use custom, unknown allocators. Of the known, non-standard allocators, 37% are from EOSIA, a WebAssembly platform for smart contracts (See Figure 3). Figuring out whether these allocators have exploitable vulnerabilities requires further research. The final heuristic concerns imported functions which are potentially harmful. They separate these functions into categories such as file I/O or code execution. While 21,2% of the binaries import at least one potentially harmful function, the overwhelming majority of these cases (in 19% of all binaries) are I/O functions (See Table 2).

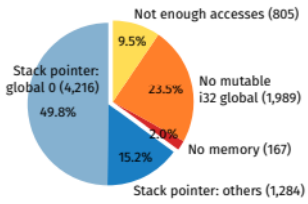

Figure 2: Result of an analysis of memory usage in real world WebAssembly binaries, taken from Hilbig et al. [4].

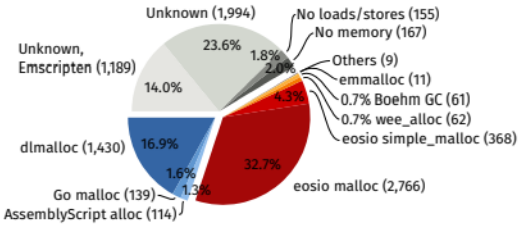

Figure 3: Result of an analysis of allocators used in real world WebAssembly binaries, taken from Hilbig et al. [4].

Table 2: Result of an analysis of potentially harmful functions imported into WebAssembly modules. Taken from Hilbig et al. [4] with some modifications.

Category	Patterns	Matches
Code execution	eval, exec, execve, emscripten_run_script	1.9%
Network access	xhr, request, http, fetch	2.0%
File I/O	file, fd, path	19.0%
Dynamic linking	dlopen, dlsym, dlclose	1.6%

6.1 Discussion

Given the results of the analysis presented in Section 6, it seems quite a lot of real-world WebAssembly binaries are potentially vulnerable. Despite this, there are no known, widely deployed exploits targeting WebAssembly binaries. Using `exploitdb` (https://www.exploit-db.com/) and searching for either the terms 'wasm' or 'webassembly' yields one and three results, respectively. The latter date back to 2018, the former to 2019. Additionally, all of them are exploits targeting the VM, not a WebAssembly binary using some malicious user input. Similarly, using google to search for the terms 'exploit webassembly', or 'webassembly binary exploit' doesn't yield any relevant exploits on the first three pages. Given that McFadden et al. [10] discussed potential WebAssembly vulnerabilities as far back as 2018, the question remains as to why there haven't been more exploits widely reported on. Especially regarding the fact that binary security remains one of the major selling points of WebAssembly, one would expect the discovery of an effective exploit in the wild to be widely shared and discussed. This suggests that the unavailability of information regarding exploits is caused by there not being any. In what follows, we suggest reasons for why that might be with suggestions regarding how to test these hypothesis where applicable.

6.1.1 The analyses are incomplete. One reason that might explain the discrepancy between the many binaries which have at least one apparent vulnerability and the lack of exploits in the wild is that the analyses we discussed are leaving out some important factor which makes exploits way less likely. For example, many of the binaries which import some dangerous function might never take any user input which could exploit them. The way to test this would be to test the collected binaries as to whether they take user input and which parts of the code are affected by this input. This could use the taint analysis discussed in Section 5.2. It would be interesting to see how many binaries use both unmanaged memory, import an unsafe function, and allow user supplied input to affect the programs behavior. This could be an interesting avenue for future work. The analysis done so far rely on static analysis and a heuristic, which isn't sophisticated compared to state-of-the-art program analysis techniques available for native binaries.

6.1.2 There are easier or more lucrative targets. Since many WebAssembly modules are only available as binaries, interpreting them well enough to find exploits might have significant overhead compared to exploiting Javascript. Additionally, it stands to reason that the vast majority of Websites still mainly rely on Javascript to handle their business logic in the front and backend. And finally, the way WebAssembly is often discussed seems to suggest using it to only speed up the critical code paths and calling into the WebAssembly code from Javascript. Although this is less true when using emscripten, which for example has functionality to write main functions or define update-render loops for games. All this seems to point towards the fact that people interested in potentially exploiting WebAssembly would find it easier and more profitable to instead focus on Javascript or native binaries. Again, this is conjecture, and hard to either verify or debunk. One could perhaps interview industry experts.

6.1.3 WebAssembly's security measures are enough. It might also be the case that the structured control flow and managed return addresses together with the sandboxed memory already lead to sufficient security. While the vulnerabilities discussed in Section 3 are valid and enable proof of concept attacks, they might still be impractical. For example, changing control flow requires that: (i) the function's address is on the unmanaged stack, (ii), some data given by the user can override that location, and (iii), another function exists with the same signature which the user can exploit. This reasoning is similar to the points given in Sections 6.1.1 and 6.1.2. Having the VM handle return addresses might simply be enough to hinder most control flow attacks. In addition, a very specific set of circumstances has to occur for a WebAssembly vulnerability to be exploitable, for example by enabling a cross site scripting attack. For example, another user (the victim) would have to trigger an exploit by calling WebAssembly with malicious input somehow supplied by the attacker. It needs be remarked that Lehmann et al. [5] indeed show an example of cross site scripting being enabled by exploiting a well-known bug CVE-2018-14550 in an image conversion library libpng. When using version 1.6.35 of this library and converting an PNM to a PNG file, there exists a buffer overflow vulnerability. If a website were to use this library to allow users to convert images, and an attacker were to supply malicious input, this overflow could be used to overwrite a string which gets later added to the DOM. Additionally, they compare the control flow defense inherent in WebAssembly to defenses deployed by native binaries, so called control-flow integrity (CFI) policies. They conclude that "Overall, WebAssembly's type checking is often less effective than modern CFI defenses available for native binaries". However, as mentioned before, diverting control flow in WebAssembly presupposes the ability to taint a very specific part of program memory. Again, testing this hypothesis doesn't seem straightforward. It would probably need both more analysis both automated and manually by experts in the field.

Given more time, it seems likely that whether WebAssembly allows for more exploits than expected will come to light as more and more Websites start using it. Wider adoption would make finding

exploits more lucrative and thus, more likely. It is plausible that the reason why there haven't been any widely talked about WebAssembly exploits (especially following the publication by Lehmann et al. [5]) is due to a mixture of the factors discussed above.

7 CONCLUSION

We introduced the WebAssembly binary format and execution environment, with a focus on possible vulnerabilities introduced by the unmanaged memory. The security measures inherent to WebAssembly such as structured control flow and managed addresses are a selling point of WebAssembly. Despite that, using unmanaged memory in combination with indirect calls and the lack of common exploit mitigation techniques, we demonstrated the existence of proof of concept attacks. These would, for example, enable cross site scripting in certain cases. Discussing a large analysis on real world WebAssembly programs which comprises over 8000 binaries, we showed that a large percentage of them have some of the vulnerabilities introduced before. This leads to an apparent contradiction when considering the fact that there haven't been any widely reported exploits unique to WebAssembly. We concluded by presenting possible reasons as to why that might be and some avenues for future research.

REFERENCES

[1] El Habib Boudjema, Sergey Verlan, Lynda Mokdad, and Christèle Faure. 2020. VYPER: Vulnerability detection in binary code. *Security and Privacy* 3, 2 (March 2020). https://doi.org/10.1002/spy2.100

[2] Patrick Cousot. 2000. Abstract Interpretation: Achievements and Perspectives. (Aug. 2000), 7.

[3] William Fu, Raymond Lin, and Daniel Inge. 2018. TaintAssembly: Taint-Based Information Flow Control Tracking for WebAssembly. *arXiv:1802.01050 [cs]* (Feb. 2018). http://arxiv.org/abs/1802.01050 arXiv: 1802.01050.

[4] Aaron Hilbig, Daniel Lehmann, and Michael Pradel. 2021. An Empirical Study of Real-World WebAssembly Binaries. (2021), 13.

[5] 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.

[6] Daniel Lehmann and Michael Pradel. 2018. Wasabi: A Framework for Dynamically Analyzing WebAssembly. (2018), 14.

[7] Jun Li, Bodong Zhao, and Chao Zhang. 2018. Fuzzing: a survey. *Cybersecurity* 1, 1 (Dec. 2018), 6. https://doi.org/10.1186/s42400-018-0002-y

[8] Runhao Li, Chen Zhang, Chao Feng, Xing Zhang, and Chaojing Tang. 2019. Locating Vulnerability in Binaries Using Deep Neural Networks. *IEEE Access* 7 (2019), 134660–134676. https://doi.org/10.1109/ACCESS.2019.2942043

[9] Pedro Daniel Rogeiro Lopes. 2021. Discovering Vulnerabilities in WebAssembly with Code Property Graphs. (2021), 31.

[10] Brian McFadden, Tyler Lukasiewicz, Jeff Dileo, and Justin Engler. 2018. Security Chasms of WASM.

[11] 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.

[12] Andreas Rossberg, Ben L. Titzer, Andreas Haas, Derek L. Schuff, Dan Gohman, Luke Wagner, Alon Zakai, J. F. Bastien, and Michael Holman. 2018. Bringing the web up to speed with WebAssembly. *Commun. ACM* 61, 12 (Nov. 2018), 107–115. https://doi.org/10.1145/3282510

[13] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Jose, CA, 138–157. https://doi.org/10.1109/SP.2016.17

[14] Pengfei Sun, Luis Garcia, Yi Han, Saman Zonouz, and Yao Zhao. 2021. Poster: Known Vulnerability Detection for WebAssembly Binaries. (April 2021), 2.

[15] Tiantian Tan, Baosheng Wang, Zhou Xu, and Yong Tang. 2018. The New Progress in the Research of Binary Vulnerability Analysis. In *Cloud Computing and Security*, Xingming Sun, Zhaoqing Pan, and Elisa Bertino (Eds.). Vol. 11064. Springer International Publishing, Cham, 265–276. https://doi.org/10.1007/978-3-030-00009-7_25 Series Title: Lecture Notes in Computer Science.