Vulnerabilities in WebAssembly: A Survey

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

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, 7 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 [16]. Since then, most major browsers such as Firefox, Chrome or Safari implement many of the proposed features. Even on the server 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. As such, this can be seen as a comparison between exploits in native binaries and WebAssembly binaries.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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" [15]. 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 a few puplications commenting on WebAssembly's lack of mitigations to common binary exploitation techniques, such as [12] or [7]. Additionally, there have been puplications researching the real-world assembly programs such as [13] or [6]. In this survey, we aim to cover the main points of these publications and comment on how well WebAssembly is protected against binary exploits by design.

2 BACKGROUND

This survey aims at researching the binary vulnerabilities of WebAssembly. To this end, it is interesting to compare how it stacks up next to classical security concerns of native binaries. There are many different binary exploits, and securing against them seems a ever changing arms race between security professionals and hackers. We will mainly concern ourselves with vulnerabilities which are due to some form of user input. 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 a deep dive into 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, which can be triggered by malicious user input.

2.1 WebAssembly: High Level Overview

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 [14], or the Background section in [7].

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 1: 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

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

char *indirect_func() {
  return str;
}

WASM_EXPORT
char *direct_func(int i) {

}
```

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

```
(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 3: The c program in Figure 2 gets compiled to this WebAssembly program (with uninteresting parts removed). Observe the function table and memory.

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.

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 [7]. As will be discussed 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 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. Figure 4 shows example code to make this more intuitive. This code prints 1,2,1,3 to the console when running it in the browser.

```
#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;
}
```

Figure 4: This Code demonstrates how WebAssembly handles function pointers. 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, 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. This is demonstrated by the code in . By copying the index of different functions directly into the function pointer, different functions get called indirectly. The function pointer supposedly takes a pointer to a char as argument. However, it is possible to use the index of a function taking an Integer as parameter.

```
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 adresses so they get put in the 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')");
```

Figure 5: Example demonstrating how function pointer type checking works.

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 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 SDL [2].

3 BINARY VULNERABILITIES OF WEBASSEMBLY PROGRAMS

In this section we will discuss the security vulnerabilities of WebAssembly as presented in [12] and [7]. [7] 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, [7] 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 Exploitation potential of unmanaged memory

[12] discuss serveral common exploit mitigation techniques which are used in binary programs. Table **??** provides a summary and shows whether these techniques are present in WebAssembly.

[7] similary 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.

4 EXPLOITING VULNERABILITIES

Having analyzed WebAssemblys main vulnerability, the unmanaged linear memory, there are several ways to exploit it. [12] 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. [7] similarly first demonstrate two write primitves. They first introduce a stack-based buffer overlfow. They also demonstrate how the allocator supplied by emscripten (called emmalloc) is succeptible to the so-called unlink exploit. This can be utilized to allow an attacker to write an arbitrary value to an arbitrary adress. However, we couldn't recreate their exploit using the current version of emscripten. Nevertheless, we will quickly summarize the presented attack primitives. Afterwards we will demonstrate our attempt to recreate these attack primitives. It must be remarked that the main source of security of WebAssembly comes from the fact that it is isolated from the surrounding environment by the Virtual Machine. 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 broser, such functions will be imported from Javscript. Examples are functions such as **eval** or functions which manipulate the DOM. **Eval** is a Javascript function which will execute any string passed to it as Javascript code. They also quickly describe how user-supplied data could lead to a stack overflow. Since WebAssembly has no guard pages, a stack overflow could overwrite data following the stack. However, no example of an exploit is given.

4.1 Overflowing the unmanaged stack

Overflowing buffers on the stack is one of the most widely used binary exploitation technique. 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 "never use gets". There are many ways to exploit a stack based buffer overflow. In native binaries without mitigations such as stack canaries, a buffer overflow can be used to overwrite the functions return adress. This can lead to executing arbitrary code with the permissions of the program. However, since the WebAssembly Virtual Machine manages return adresses seperately, one could assume that stack based buffer overflows can't be as easily abused. However, if the function has non-scalar data, it will have memory on the unmanaged stack. In fact, since there exist no stack canaries in WebAssembly, a buffer overflow can overwrite any memory before the functions 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. [7] have serveral examples to demonstrate this vulnerability. 6 shows an example inspired by them. The main function generated a string which gets added to the dom. Before then, 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.

```c
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;
}
```

Figure 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 4.1? Since the return adresses are managed by the Virtual machine, the answer seems to be no. However, it is possible

Exploit mitigation	Protect against	Effect in Native Binaries	WebAssembly
Address Space Layout Randomization (ASLR)	Attacks on Control-Flow	Randomizes the base adress of an executable as well as the heap and stack adresses and adresses 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 Adresses, many exploits that rely on changing control flow are infeasible. In addition, since WebAssembly only provides 32-bit adresses, it is assumed to have to little entropy for ASLR to be effective. Also, since functions are directly accessed by indices instead of memory adresses, ASLR could not be used to obfuscate function adresses.
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 adress. This enables the program to detect and handle these overflows	Stack canaries don't exist in WebAssembly, since the return adresses 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 deploy with their own smaller allocator which might have vulnerabilities.
Data Execution Prevention (DEP)	Code injection	This mitigation is part of a family of mitigations which modify memory pages to only allow certain behaviour. These can 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 seperation between code and data. Hence, DEP is not needed.

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

to overwrite indirect function calls. As discussed in 2.6, WebAssembly references functions by an 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 arguments and return type as the newly called function.

Also, if we wish to influence the parameters passed to the newly passed 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. [12] demonstrate a proof of concept of using a overflow to manipulate control flow. We couldn't recreate their exact example, however a slight modification did work. The example in Figure 8 works on Firefox version 88.0.1 64-Bit when compiled with emscripten version 1.39.16.

The imagined scenario is that of a legacy application being ported to WebAssembly. It is possible to send messages and handle them using different functions. A spcially crafted payload can be used to overflow into the **msg_len** and **out** member fields, changing the program behaviour. Again, it uses 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 using this to steal cookies, etc. The above example is particularly interesting since it could only work in WebAssembly and not in other native programs.

5 BINARY VULNERABILITY DISCOVERY

5.1 Native Binaries

Trying to find out whether a program will crash is equivalent to the Halting problem. Thus, 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. See for example [19] for a more thorough review.

Techniques to discover binary vulnerabilities can roughly be divided into two categories: Static and dynamic analysis techniques [17]. 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 of the problems is deciding how to trade-off false positive edges. This is especially the case when dealing with indirect

```
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 so it gets added to the 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;
}
```

Figure 7: This code demonstrates a proof of concept of an exploit using indirect calls to influence control flow and implement cross site scripting.

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 and many false positives.

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

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

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 [9]. These aim to generate inputs that trigger interesting behavior in the target program. Creating a Fuzzer which generate 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

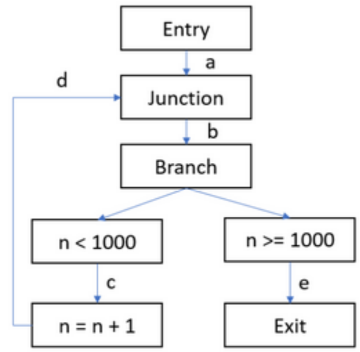

Figure 8: A Control Flow Graph derived from the Pseudocode in listing 1

code are influenced (tainted) by user input. These analyses require some understanding of the source program. Thus, 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 method of based on programming language semantics and their fixed points [4]. Intuitively, the program is run in an abstract way where instead of using explicit values, it is run with all possible values of a given type. For example, instead of running a program using integers as input, the values are replaced by intervals. These Operands 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 in [17]. It provides tools for loading binaries, computing CFGs, etc. [3] builds on angr to implement VYPER, a tool which utilizes concolic execution of the binary target. This is supposed 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 [10].

5.2 WebAssembly

There are several tools and approaches to specifically analyze WebAssembly Binaries to locate possible exploits. One proposed tool, Wasmati, is introduced in [11]. It aims to utilize Code Property Graphs. While promising, it has yet to be implemented and tested. TaintAssembly, introduced in [5] 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. [18] describes 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 [8] 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. Thus, for example, there is one hook for all function calls. If implemented, a user-implemented callback is run on WebAssembly function calls while the function name and other information gets passed to the callback. Only those hooks which are implemented in the analysis get put into the binary to improve performance. Wasabi is the basis for the static analysis tools presented in [7] and [6]. It is also used in [18]. Since these will form the basis for the analysis of real-world binaries which will be presented in REF REAL WORLD, we will go into more depths on how they are implemented. As mentioned above, a static analysis tool is presented which uses heuristics to try and find possible vulnerabilities. Using the Wasabi Rust implementation as a library, the stack pointer 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 according to manual verification. To identify possible vulnerabilities the tool looks for either of the following: use of the unmanaged stack as introduced in 2.3, unsafe memory allocators such as emmalloc 3 or access to potentially dangerous imported functions such as **eval**. The results will be discussed in ANALYSIS DISCUSSION

6 REAL-WORLD WEBASSEMBLY BINARIES

To date, there exist three major analyses of WebAssembly binaries "in the wild": One in 2019 [13] focused on the way WebAssembly is used by the most popular websites. [7] uses the WebAssembly static analysis tool explained in 5.2 on real-world WebAssembly to ascertain whether the vulnerabilities they uncovered exist in deployed code. Finally, in 2021 [6] ran the thus far largest analysis combining the methods of both [13] and [7]. This analysis comprises 8,461 unique WebAssembly binaries. Their results are: 65% use unmanaged memory.

REFERENCES

[1] [n.d.]. . https://emscripten.org/

[2] [n.d.]. . https://www.libsdl.org/index.php

[3] 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

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

[5] 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.

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

[7] 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.

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

[9] 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

[10] 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

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

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

[13] 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.

[14] Andreas Rossberg. 2021. *WebAssembly Specification.* https://webassembly.github.io/spec/core/

[15] 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

[16] Nick Schonning. 2021. *Loading and running WebAssembly code.* https://webassembly.org/roadmap/

[17] 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

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

[19] 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.