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, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

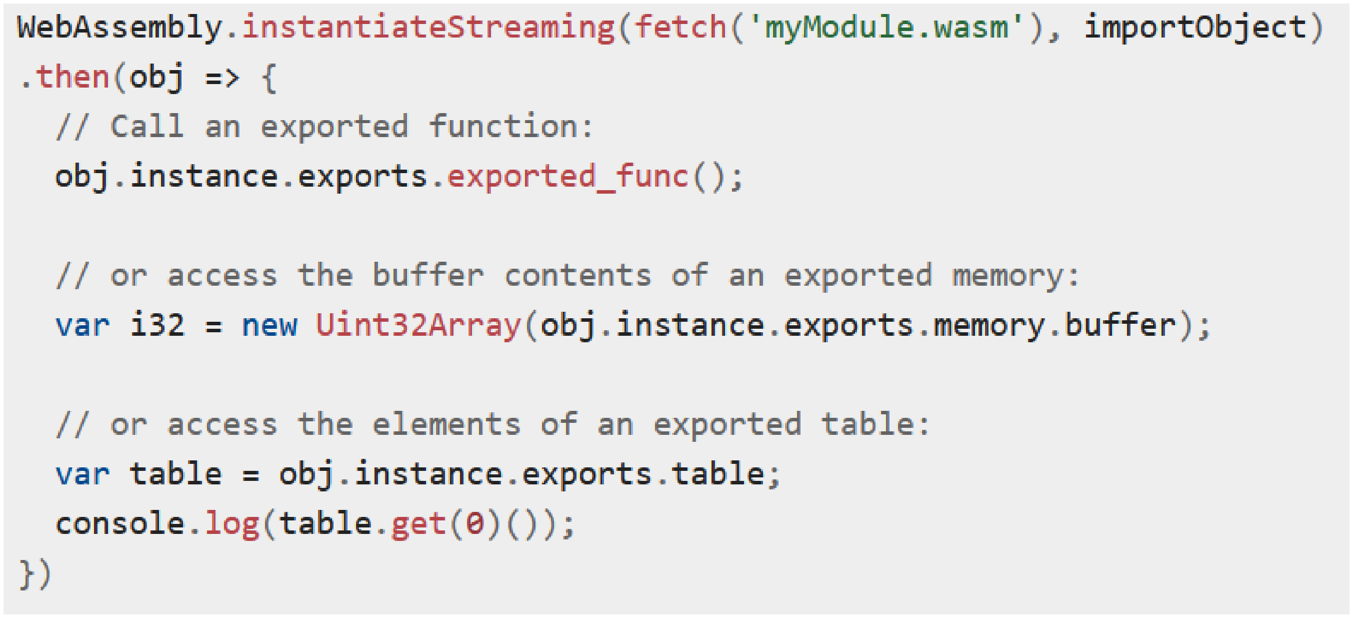
1 INTRODUCTION

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 [6]. 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 [3] or [2]. Additionally, there have been puplications researching the use and existence of security vulnerabilities in real-world assembly programs such as [1] or [4].

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 [5].

2.0.1 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 **??**. WebAssembly functionality is then accessed by calling functions which are exported by the WebAssembly module from Javascript. 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).

2.0.2 Modules. At the highest level, WebAssembly programs are organized into Modules.

REFERENCES

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

[2] 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.

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

[4] 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.

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

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