SOPHOS
Cybersecurity made simple.

Understanding WebAssembly

An in-depth peek into the VM running in modern web browsers.

Author: Christophe Alladoum, Security Researcher, SophosLabs

This whitepaper aims to provide a global understanding of WebAssemble – the file format, the instruction set, and also to analyze it from an offensive perspective to try and determine if and how this new format changes the attack surface on modern web browsers.

Table of Contents

SOPHOS

Introduction

In March 2017, the World Wide Web Consortium (W3C) published the first stable specification of WebAssembly (WASM), a new format defined mostly (but not solely) for web environment. The idea behind WebAssembly is to provide an environment for client-side application to be executed with performance as close to native as possible.

Today, all major web browsers (Chrome, Edge, Safari, Firefox) support this format. As a matter of fact, according to CanIUse.com, approximately 75% of web browsers, all platforms included (PC/Mac, phones, tablets, etc.) are WebAssembly-capable. The reason for such enthusiasm stems from WASM's very aggressive computation performance, which allows running real-time applications or video games entirely from the web browser, which JavaScript could never reach.

WebAssembly was built with several security considerations, that makes it impervious to trivial attacks (like control flow hijack via return address in stack overwrite), or out-of-bounds memory dereferencing.

Despite its presence in every major browser, WebAssembly is not very well-known by developer and security communities. Therefore, in this paper, we will try to fill that gap by examining how WebAssembly works, its file format, and instruction set. After detailing its internal structure when loaded by web browsers, we'll focus on the security of this technology, and covers some bugs present and past in web browsers that leveraged WebAssembly to hijack the execution flow. We will also discuss some potential existing risks in this technology, along with some paths for remediation.

WebAssembly

WebAssembly (WASM)1 is a technology developed co-jointly by Google, Microsoft, and Mozilla – the result of two years of work. The focus was to provide a high-performance environment for web applications to run on, all in a contained, browser-controlled environment. It is an open W3C standard, and the first stable release was published in March 2017, as a Minimum Viable Product.

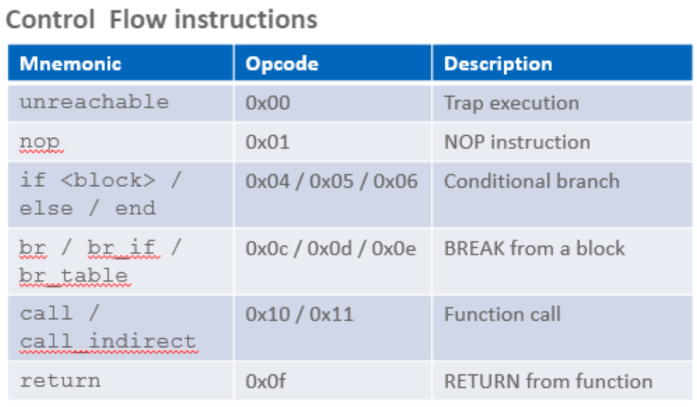
Minimum Viable Product

The Minimum Viable Product[2] provides a minimum yet complete WebAssembly environment specification that also leaves room for future improvements within the technology itself. It is the first version of the WebAssembly specification, and the only one modern Web browsers implement. Among other things, the MVP declares the design of WebAssembly, including primarily:

- The virtual machine design, and module structures
- The instruction set
- The file format
- The binary encoding

Instruction Set

The WASM MVP defines a total of 172 instructions. The syntax is very RISC-like and allows you to perform some logic and arithmetic operations (including on float IEEE 754 – 32 and 64 bits), load (respectively store) from (resp. to) memory local or global, along with control flow instructions (conditional branch, function call and return, conditional looping, switches). A comprehensive and exhaustive list of all WASM instructions and their syntaxes can be found on GitHub[3].



Control Flow instructions

Mnemonic	Opcode	Description
`unreachable`	0x00	Trap execution
`nop`	0x01	NOP instruction
`if <block> / else / end`	0x04 / 0x05 / 0x06	Conditional branch
`br / br_if / br_table`	0x0c / 0x0d / 0x0e	BREAK from a block
`call / call_indirect`	0x10 / 0x11	Function call
`return`	0x0f	RETURN from function

Figure 1: Examples of WASM control flow instructions

1 https://webassembly.org
2 https://github.com/WebAssembly/design/blob/master/MVP.md
3 https://github.com/sunfishcode/wasm-reference-manual/blob/master/WebAssembly.md#instructions

One noticeable point lays in the fact that there is no `syscall`-like instruction: (to the exception of the `grow_memory` and `current_memory` instructions, but those instructions have no effect in the MVP 1.0). Therefore, this is a strong indicator that WebAssembly was designed for computational purpose of web applications, leaving JavaScript to handle the interactive part.

WebAssembly Virtual Machine

One key design component of the WASM is that it is upward-growing stack-based, very much like Java or Python: simply put, a stack-based VM will rely on the stack for all its operations (function calls, conditional branches, etc.), to the difference of register-based VM, which will define a set of registers for this. A simple `add 8,16` instruction in WASM will look as such:

```
CODE:0478                    i32.const   #8

CODE:047A                    i32.const   #16

CODE:047C                    i32.add
```

With the result of this addition pushed onto the stack.

The VM uses a 32-bit flat address space, and uses a page granularity of 65536 bytes (64KB), unlike typical operating system pages that are 4KB. A crucial difference with other ABIs (such as Intel, ARM, etc.) is the fact that WASM doesn't manipulate pointers. Consequently, there is no `call FunctionAddress` instruction in WASM. Instead, WASM will define tables with indexed entries, and calling a function means calling the index to this function. When hitting a call instruction, such design allows the loader to check that there is an existing indexed function signature. As such this kills all types of control flow redirection attacks (arbitrary function call, JOP/ROP), it is only possible to invoke existing functions, and follow their execution flow.

File format

WebAssembly file format (`.wasm`) aims to be simple and extensible. Each WebAssembly is referred to individually as a **module**. Each module must have a valid header, followed by 0 or more section as detailed below.

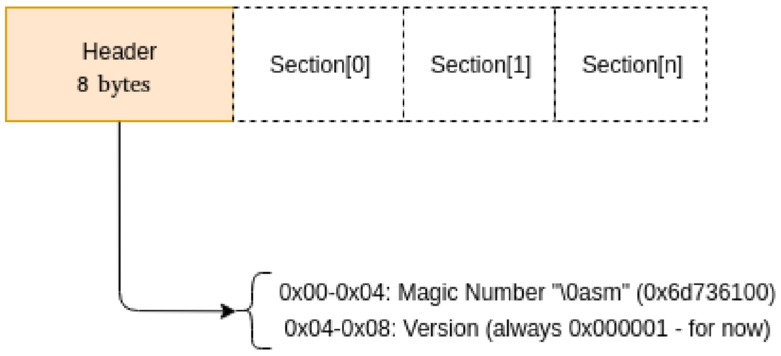

Header
8 bytes
Section[0]
Section[1]
Section[n]
0x00-0x04: Magic Number "\0asm" (0x6d736100)
0x04-0x08: Version (always 0x000001 - for now)

Figure 2: General structure of a WASM file.

Header

The WebAssembly header is a static 8-byte field that starts with the magic DWORD 0x00617364 (or `\0asm` in ASCII) followed by the compatibility version of the module encoded as a DWORD. Today, only the value 1 exists and is valid.

Sections

A section comprises a header and payload part: the header holds a unique identifier, the section **code**, which is a 1-byte integer that states the nature of the current section, and dictates the structure of the payload. According to the specification, a WASM module cannot have more than one section with the same code, to the exception of the Custom section identifier (value 0). Such sections are uniquely identified by the section name, which must be explicitly present in the section header.

Currently sections are heavily encoded using Variable-Quantity Length[4] format, which helps compressing partially the volume. Future specification may introduce section compression mechanism.

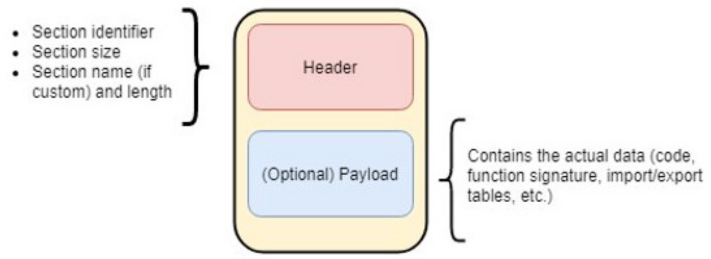

Section identifier
Section size
Section name (if custom) and length
Header
(Optional) Payload
Contains the actual data (code, function signature, import/export tables, etc.)

Figure 3: WASM section structure.

The MVP documents nine non-custom sections, including:

- `Function` section: defines the signature of all the functions in the current WASM module. A signature consists in the number and type of arguments (0 or more), along with the number and type of return value (at most 1). The signatures are stored in the *Function Index Space.*
- `Code` section defines the bytecode of all the functions, whose signatures were defined in the Function section. `Code` and `Function` sections are intrinsically linked.
- Global section defines all the global variables of the module, which are stored in the *Global Index Space*.
- `Export` (resp. Import) section declares all objects (functions, global, memory) to be exported (resp. imported). Exported objects are referenced by the indexes in their index space.
- `Start` section refers to a function index that should be called after the WASM module was instantiated.
- `Memory` section corresponds to the declaration of the internal linear memory.

4 https://en.wikipedia.org/wiki/Variable-length_quantity

- A basic WASM module would look be coded as follow:

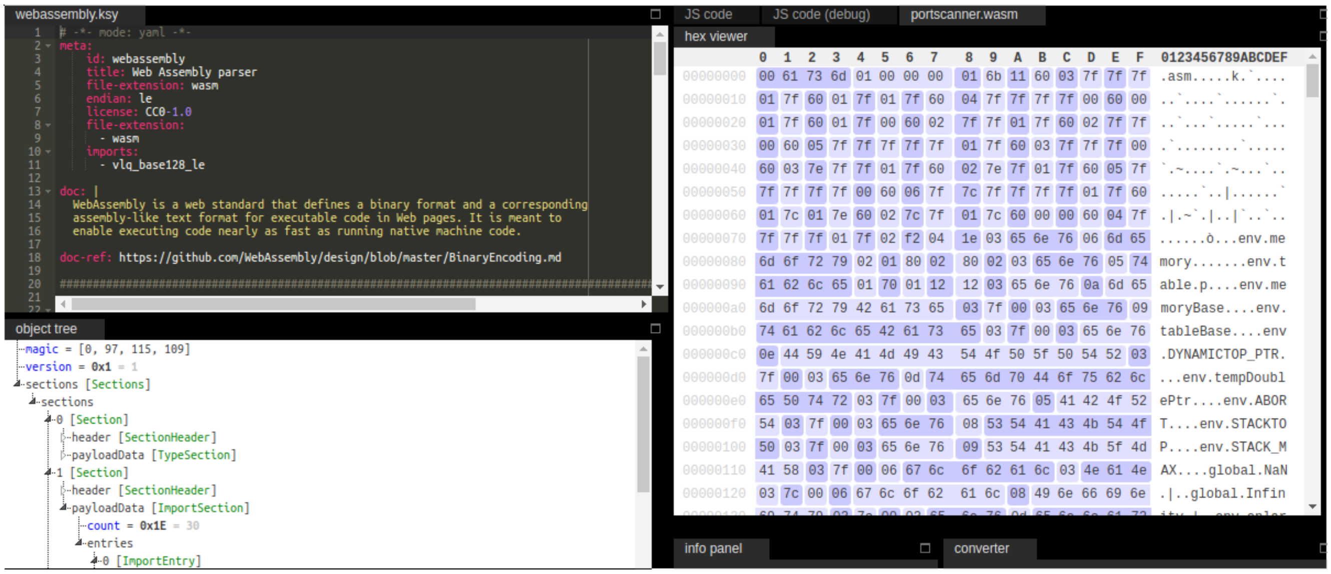
```
(module
  (import stdlib print (func $print (param i32 i32)))
  (import js memory (memory 20)) ;; initial allocation of 20 pages
  (data (i32.const 0) Hello world!”) ;; store string in data segment at
offset=0 (length=strlen(“Hello world!”)=12)

  (func (export main”)
    i32.const 0  ;; offset=0
    i32.const 12 ;; length=12
    call $print
  )
)
```

Which once compiled with wat2wasm from WebAssembly Binary Toolkit[5] would produce the following binary file:

```
00000000: 0061 736d 0100 0000 0109 0260 027f 7f00  .asm.......`....
00000010: 6000 0002 1d02 0673 7464 6c69 6205 7072  ......stdlib.pr
00000020: 696e 7400 0002 6a73 066d 656d 6f72 7902  int...js.memory.
00000030: 0014 0302 0101 0708 0104 6d61 696e 0001  ..........main..
00000040: 0a0a 0108 0041 0041 0f10 000b 0b12 0100  .....A.A........
00000050: 4100 0b0c 4865 6c6c 6f20 576f 726c 6421  A...Hello World!
```

A Kaitai[6]-based parser for the WebAssembly format was published on the GitHub repository (you'll find a link in the appendix) to help improving the visualization of all the different sections of WASM. An IDA Pro loader and processor can be found in the same repository to assist analysts in the process of auditing potentially hostile WASM code.

5 https://github.com/WebAssembly/wabt
6 https://kaitai.io



```
# -*- mode: yaml -*-
meta:
    id: webassembly
    title: Web Assembly parser
    file-extension: wasm
    endian: le
    license: CC0-1.0
    file-extension:
      - wasm
    imports:
      - vlq_base128_le

doc: |
  WebAssembly is a web standard that defines a binary format and a corresponding
  assembly-like text format for executable code in Web pages. It is meant to
  enable executing code nearly as fast as running native machine code.

doc-ref: https://github.com/WebAssembly/design/blob/master/BinaryEncoding.md

##########################################################################################
```

```
magic = [0, 97, 115, 109]
version = 0x1 = 1
sections [Sections]
  sections
    0 [Section]
      header [SectionHeader]
      payloadData [TypeSection]
    1 [Section]
      header [SectionHeader]
      payloadData [ImportSection]
        count = 0x1E = 30
        entries
          0 [ImportEntry]
```

```
          0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F  0123456789ABCDEF
00000000  00 61 73 6d 01 00 00 00  01 6b 11 60 03 7f 7f 7f  .asm.....k.`....
00000010  01 7f 60 01 7f 01 7f 60  04 7f 7f 7f 7f 00 60 00  ..`....`......`.
00000020  01 7f 60 01 7f 00 60 02  7f 7f 01 7f 60 02 7f 7f  ..`...`.....`...
00000030  00 60 05 7f 7f 7f 7f 7f  01 7f 60 03 7f 7f 7f 00  .`........`.....
00000040  60 03 7e 7f 7f 01 7f 60  02 7e 7f 01 7f 60 05 7f  `.~....`.~...`..
00000050  7f 7f 7f 7f 00 60 06 7f  7c 7f 7f 7f 7f 01 7f 60  .....`..|......`
00000060  01 7c 01 7e 60 02 7c 7f  01 7c 60 00 00 60 04 7f  .|.~`.|..|`..`..
00000070  7f 7f 7f 01 7f 02 f2 04  1e 03 65 6e 76 06 6d 65  ......ò...env.me
00000080  6d 6f 72 79 02 01 80 02  80 02 03 65 6e 76 05 74  mory.......env.t
00000090  61 62 6c 65 01 70 01 12  12 03 65 6e 76 0a 6d 65  able.p....env.me
000000a0  6d 6f 72 79 42 61 73 65  03 7f 00 03 65 6e 76 09  moryBase....env.
000000b0  74 61 62 6c 65 42 61 73  65 03 7f 00 03 65 6e 76  tableBase....env
000000c0  0e 44 59 4e 41 4d 49 43  54 4f 50 5f 50 54 52 03  .DYNAMICTOP_PTR.
000000d0  7f 00 03 65 6e 76 0d 74  65 6d 70 44 6f 75 62 6c  ...env.tempDoubl
000000e0  65 50 74 72 03 7f 00 03  65 6e 76 05 41 42 4f 52  ePtr....env.ABOR
000000f0  54 03 7f 00 03 65 6e 76  08 53 54 41 43 4b 54 4f  T....env.STACKTO
00000100  50 03 7f 00 03 65 6e 76  09 53 54 41 43 4b 5f 4d  P....env.STACK_M
00000110  41 58 03 7f 00 06 67 6c  6f 62 61 6c 03 4e 61 4e  AX....global.NaN
00000120  03 7c 00 06 67 6c 6f 62  61 6c 08 49 6e 66 69 6e  .|..global.Infin
```

Figure 4: WASM parser for Kaitai.

Security of WebAssembly

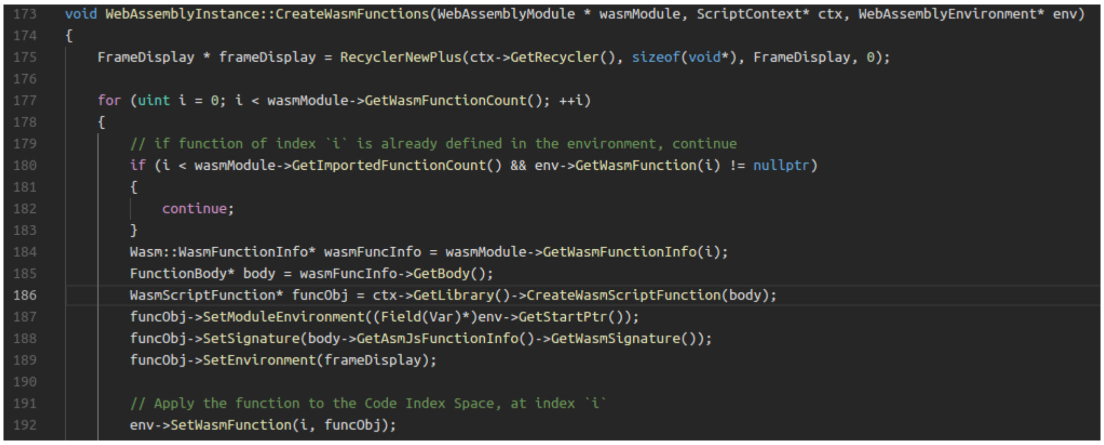
The MVP dedicates an entire section[7] to the security consideration around WASM. It should prevent malicious modules from escaping the sandboxed environment from where it executes. In addition, two WASM modules loaded are viewed as separate (in the context of the process memory layout), and can only interact when explicitly allowed by developers.

Also, because each module runs within the context of the DOM, several HTTP properties must be assured, such as the respect of the Same-Origin Policy (SOP). However, today the Content Security Policy (CSP) is not properly considering the execution of WASM modules: it assumes that JavaScript is responsible of loading and running WASM code, and therefore leaving the CSP filtering at the JavaScript level. This might become problematic in the near future, as the next releases of the specification will allow HTML code to invoke WASM directly via a dedicated `<script>` type (more similar examples will be detailed in the Future features sub-section below).

The WASM VM provides some intrinsic properties to avoid common low-level programming mistakes to be exploitable:

- The Code (among other sections) is immutable: it is not possible to create new code from inside the sandbox
- Some Control-Flow Integrity (CFI) is provided by the fact that pointers have no significant meaning in WASM (i.e. the memory cannot be dereferenced). As a reminder, this is provided by the Index Space mechanism, that immediately (i.e. from the initialization) allows detecting access to incorrect offsets.

7 https://github.com/WebAssembly/design/blob/master/Security.md



```
void WebAssemblyInstance::CreateWasmFunctions(WebAssemblyModule * wasmModule, ScriptContext* ctx, WebAssemblyEnvironment* env)
{
    FrameDisplay * frameDisplay = RecyclerNewPlus(ctx->GetRecycler(), sizeof(void*), FrameDisplay, 0);

    for (uint i = 0; i < wasmModule->GetWasmFunctionCount(); ++i)
    {
        // if function of index `i` is already defined in the environment, continue
        if (i < wasmModule->GetImportedFunctionCount() && env->GetWasmFunction(i) != nullptr)
        {
            continue;
        }
        Wasm::WasmFunctionInfo* wasmFuncInfo = wasmModule->GetWasmFunctionInfo(i);
        FunctionBody* body = wasmFuncInfo->GetBody();
        WasmScriptFunction* funcObj = ctx->GetLibrary()->CreateWasmScriptFunction(body);
        funcObj->SetModuleEnvironment((Field(Var)*)env->GetStartPtr());
        funcObj->SetSignature(body->GetAsmJsFunctionInfo()->GetWasmSignature());
        funcObj->SetEnvironment(frameDisplay);

        // Apply the function to the Code Index Space, at index `i`
        env->SetWasmFunction(i, funcObj);
```

Figure 5: Extract of ChakraCore source code: initialization of WASM functions.

- A separate stack is used to store the return pointer (and some additional information). Even if stack smashing attacks are achievable, only application data will be overwritten.

What this means is that although WebAssembly cannot protect against poorly written code (security-wise), exploiting those weaknesses would not at all compromise the web browser's security. All those built-in features provide strong protections natively from memory corruption attacks:

- A stack overflow in the code would not lead to the corruption of the return address (and hijack of the execution flow).
- Traditional ROP/JOP exploitation technique would not work either.
- Out-of-bounds accesses can be detected and trapped at runtime, triggering an OOB memory error in JavaScript.

Vulnerabilities using WASM implementations

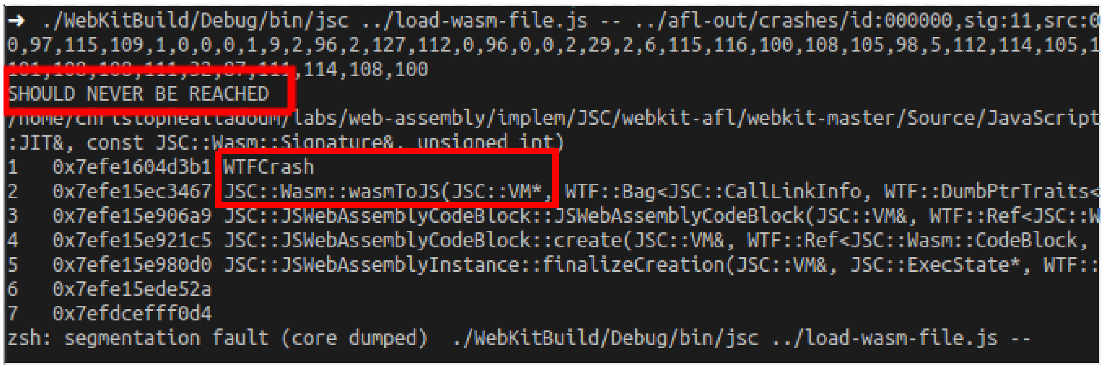
In the last few years, only a few bugs were found leveraging WebAssembly, and interestingly they all target the implementations (in V8, ChakraCore, or JSC), not the protocol. Among those vulnerabilities, we can quote:

- CVE-2017-5116
- Chrome-766253
- Project Zero P0-1522
- Project Zero P0-1545
- Project Zero P0-1526

- They are however very interesting to examine on technical perspective, as their exploitation is not exactly trivial, and diverge from traditional web browsers vulnerabilities (type confusion, Use-after-Free, Double-Free, etc.) CVE-2017-5116[8] is a good example of this: as part of their exploit chain for the Google Pixel, Qihoo 360 discovered and exploited the fact that `WebAssembly.Module` were backed by a JavaScript `SharedArrayBuffer`, which allows concurrent access to a specific memory location by 2 (or more) threads. This resulted in a TOCTOU race condition, where it was possible to change 1 byte of code after the WASM module check but before the code gets JIT-ed. As a result, it was possible to use the WASM code to read and write outside of the WASM environment, bypassing ASLR.

Although this CVE offers a great exploitation example of how to leverage WASM for web browser exploitation, modern browsers cannot be impacted from this bug or other similar because properties such as `SharedArrayBuffer` are disabled by default (as part of Spectre[9] mitigations), killing any sort of race conditions. In addition, WASM implementations in all major browsers are actively tested (through code instrumentation based fuzzing, like AFL [10]or LibFuzzer[11]).

During our research, no major bug was found within the WASM implementations of the major Web browsers, assessed through both static code reviews and fuzzing. Since WebAssembly provides a specification that defines strictly the behavior, the attack surface is rather limited. Although some implementations (such as V8 or Chakra) performs strict checks during the module initialization, emitting a JavaScript exception upon failure, JavaScriptCore prefers a more drastic approach by simply killing the contained process (by dereferencing an invalid address) if ever faulty code is found:

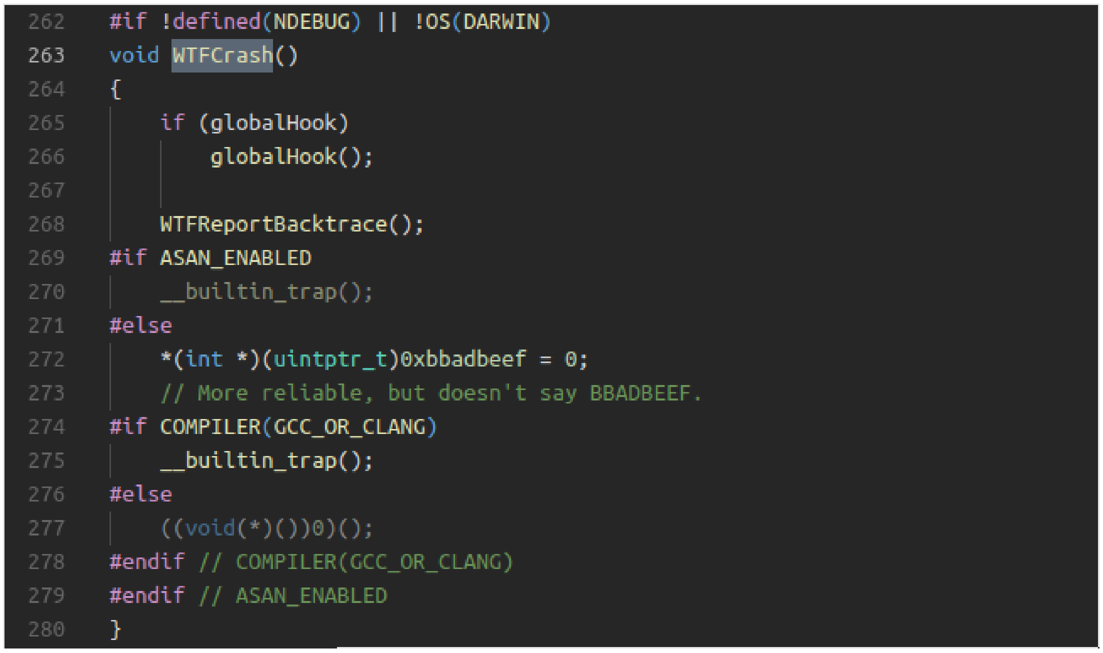


```
➜ ./WebKitBuild/Debug/bin/jsc ../load-wasm-file.js -- ../afl-out/crashes/id:000000,sig:11,src:0
0,97,115,109,1,0,0,0,1,9,2,96,2,127,112,0,96,0,0,2,29,2,6,115,116,100,108,105,98,5,112,114,105,1
101,108,108,111,32,87,111,114,108,100
SHOULD NEVER BE REACHED
/home/christophealladoum/labs/web-assembly/implem/JSC/webkit-afl/webkit-master/Source/JavaScript
:JIT&, const JSC::Wasm::Signature&, unsigned int)
1   0x7efe1604d3b1 WTFCrash
2   0x7efe15ec3467 JSC::Wasm::wasmToJS(JSC::VM*, WTF::Bag<JSC::CallLinkInfo, WTF::DumbPtrTraits<
3   0x7efe15e906a9 JSC::JSWebAssemblyCodeBlock::JSWebAssemblyCodeBlock(JSC::VM&, WTF::Ref<JSC::W
4   0x7efe15e921c5 JSC::JSWebAssemblyCodeBlock::create(JSC::VM&, WTF::Ref<JSC::Wasm::CodeBlock,
5   0x7efe15e980d0 JSC::JSWebAssemblyInstance::finalizeCreation(JSC::VM&, JSC::ExecState*, WTF::
6   0x7efe15ede52a
7   0x7efdcefff0d4
zsh: segmentation fault (core dumped)  ./WebKitBuild/Debug/bin/jsc ../load-wasm-file.js --
```

Figure 6: Crash in JSC found using AFL.

Although extreme, this is another indicator that the code was designed with security in mind, by not letting an attacker take advantage of an uncertain memory state.

8 https://android-developers.googleblog.com/2018/01/android-security-ecosystem-investments.html
9 https://meltdownattack.com
10 http://lcamtuf.coredump.cx/afl/
11 https://llvm.org/docs/LibFuzzer.html



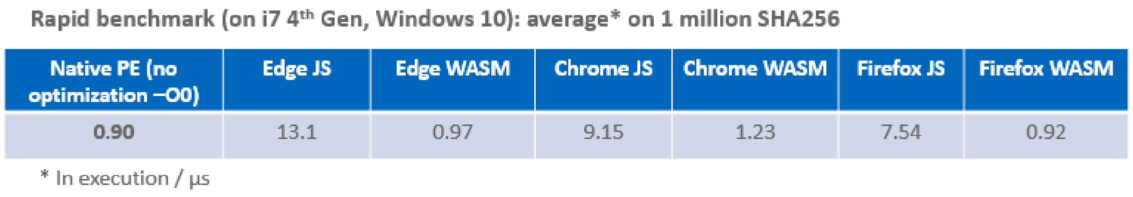
```
#if !defined(NDEBUG) || !OS(DARWIN)
void WTFCrash()
{
    if (globalHook)
        globalHook();

    WTFReportBacktrace();
#if ASAN_ENABLED
    __builtin_trap();
#else
    *(int *)(uintptr_t)0xbbadbeef = 0;
    // More reliable, but doesn't say BBADBEEF.
#if COMPILER(GCC_OR_CLANG)
    __builtin_trap();
#else
    ((void(*)())0)();
#endif // COMPILER(GCC_OR_CLANG)
#endif // ASAN_ENABLED
}
```

Figure 7: Extract from JavaScriptCore showing the invalid dereference.

Offensive WASM

Cryptomining

As part of web browsers, the performance of WASM makes it a target of choice for web-based cryptomining campaign. Internal tests have shown that (depending on the specific implementation) WASM code could execute pure computation operations (in our tests – SHA1 and SHA256) almost as fast as non-optimized code.



Rapid benchmark (on i7 4th Gen, Windows 10): average* on 1 million SHA256

Native PE (no optimization –O0)	Edge JS	Edge WASM	Chrome JS	Chrome WASM	Firefox JS	Firefox WASM
0.90	13.1	0.97	9.15	1.23	7.54	0.92

* In execution / µs

Some open source cryptominers can already be downloaded from GitHub[12][13], and it is very likely to see more in the near future.

Unfortunately differentiating two WASM modules (say for instance a cryptominer and a real-time application) is not trivial and could be subject to false positive: the produced binary code could have been obfuscated by a compiler, making it hard to retrieve static information; and the lack of syscall prevents a fully dynamic detection. An ultimate resort would be to disable WASM altogether, but to this day, only Chrome and Firefox allow this.

12 https://github.com/burland/cryptonight
13 http://www.wasmrocks.com/topic/196/javascript-emscripten-bitcoin-miner

Evasion

Another offensive use case for WebAssembly could be for defense mechanism evasion, such as WAFs or certain IDs, which use pattern-based engines. Using the C language and a compiler such as EmScripten, it becomes trivial to transform and obfuscate a JavaScript script to be `eval`-ed. It should however be noted that by specification (and confirmed on all implementations tested), the WASM modules are subject to the same Content Security Policies (CSP), and to the SOP: therefore, this risk could be mitigated by setting finely-tuned CSP rules.

Future features

WASM MVP was meant to be a first stable release of the new WebAssembly ecosystem, and as such, was built to be extensible to new features brought by newer versions of the specification. Some new improvements are already being discussed[14] and beta-tested in the bleeding-edge versions of browsers.

Among the most interesting improvements, we can find:

- On-demand memory allocation: implementing a `mmap()`-like operation (`munmap()` too). This would also introduce to WASM the concept for permission (`mprotect()`-like).
- 64-bit BigInt support.
- SIMD (Single Input Multiple Output) for 128-bit floats
- Multi-threading
- Mutable Import/Export global
- ECMAScript module support, to run WebAssembly module directly from HTML `<script>` tags.

This brings a whole new set of questions to the WebAssembly memory model which may introduce new bugs.

14 https://github.com/WebAssembly/design/blob/master/FutureFeatures.md

Conclusion

Despite a slow adoption, WebAssembly may become very popular in future web applications as it fills one major gap JavaScript never could: providing a simple but efficient platform for client-side computations. Its first specification, the MVP, provides a complete environment for applications to run on, and can be easily be extended to support more features. Security was not left out as the WASM format prevents, by essence, certain types of memory corruption or control flow hijacking attacks. In addition, the WASM modules implemented in every modern browser were tested and only few minor bugs were found during this research.

Nevertheless, it should be noted that WASM adds a non-negligible level of complexity for existing protection software (WAF, web browser dynamic analysis tool) by enabling innovative ways of code obfuscation, making hostile client-side code harder to detect.

Possible mitigations for that situation must come at two levels: first, the creation of an offline VM[15] to run the potentially malicious WASM code in a contained and confined environment. Second, the current tools (IDA, WABT, EmScripten) must be improved to assist in the static analysis process of WASM modules.

Appendix

Additional links:

- https://github.com/Sophos/WebAssembly/
- https://mbebenita.github.io/WasmExplorer/
- https://webassembly.studio/
- https://wasdk.github.io/WasmFiddle/
- https://github.com/kripken/emscripten
- https://github.com/WebAssembly/wabt

15 Basic tests have shown that such analysis can be possible by patching the WAVM project.