Web-(Dis)Assembly

An in-depth peek inside the VM running in your Web browser

Christophe Alladoum

Offensive Security Research - SophosLabs

Shakacon X - 07/18

SOPHOS

Introduction

SOPHOS

Preamble

- Who?
  - Chris (@_hugsy_ on IRC / Twitter)
  - Security researcher for SophosLabs
  - CTF, low-level stuff addict, tool builder, software breaker

- Why?
  - Asymmetry of “*I’ve heard of WebAssembly*” vs “*I know what WebAssembly is*”
  - Huge new attack surface inside all browsers (mobile devices included)
  - Is it worth checking out ?
  - Can I detect / trace / debug it ?
  - Can I cover everything in 45 minutes ?

Agenda

- *Introduction*
- Brief history of Web Assembly (WASM)
- WASM Minimum Viable Product (MVP), 1.0
- WASM attack surface
  - Implementation in Web browsers
  - Past bugs
- The future of WASM
- Conclusion & Q&A

Scope

- WebAssembly: what we’ll cover
  - Specification
    - File format
    - Instruction set
  - Focus on Web Browsers
    - Interaction with DOM / JS Engine
    - Attack surface / Fuzzing
    - Past vulnerabilities leveraging WASM

- What we won’t cover (in details)
  - Other use of WASM outside the Web world

NaCl ?

Asm.Js ?

WASM ?

Once upon a time…

- There is more to life than JavaScript !

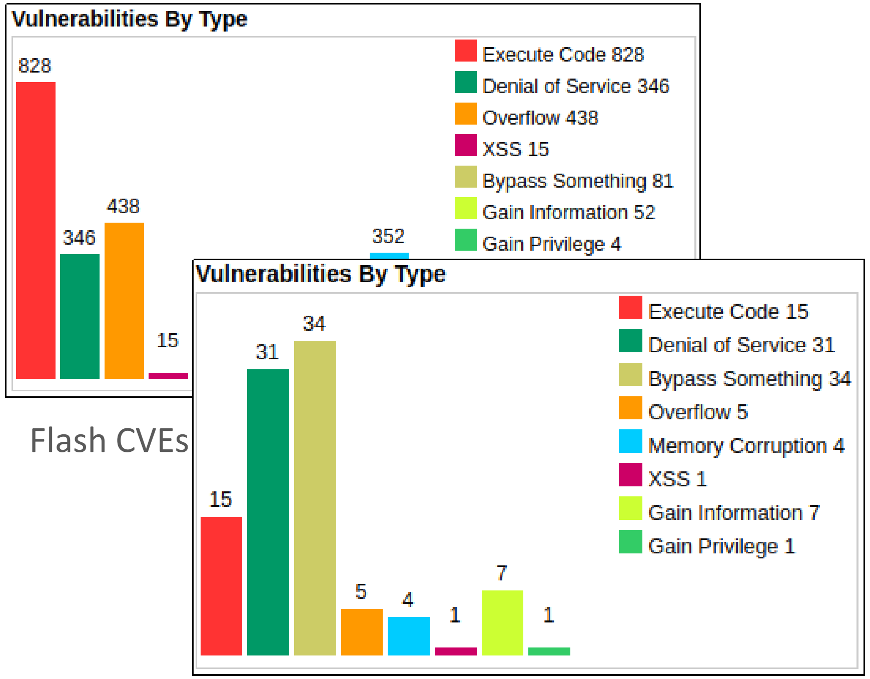
- Wanting to execute fast custom code from Web apps has been there for ages
  - Most commonly used: ActiveX, Flash, Java
  - Also emerged some terrible ideas: SilverLight, ShockWave

Once upon a time…

- There is more to life than JavaScript !

- Wanting to execute fast custom code from Web apps has been there for ages
  - Most commonly used: ActiveX, Flash, Java
  - Also emerged some terrible ideas: SilverLight, ShockWave

- But
  - Badly insecure
  - Tough to secure/control at runtime by the browser
  - Proprietary solutions
    - Never standardized



Flash CVEs

Java CVEs (source: cvedetails.com)

NaCl

- Google Native Client (2011)
  - Designed to execute native code (x86/x64, ARM) in a sandbox
  - Close to native performance
  - 3D acceleration
  - Debuggable using GDB-Remote

NaCl

- Google Native Client (2011)
  - Designed to execute native code (x86/x64, ARM) in a sandbox
  - Close to native performance
  - 3D acceleration
  - Debuggable using GDB-Remote

- But
  - Not a standard
  - Google Chrome {Browser,OS} specific
    - Doesn’t apply to other browsers

Asm.Js

- Mozilla Asm.Js (2013) – *“use strict;”*
  - Specification defining a strict subset of JavaScript
    - Source to source code translation (from C)
    - Ahead of time optimization
  - EmScripten suite created to “compile” C/C++ code to Asm.JS code
  - Can be up to 2x faster than traditional JavaScript
- Mission accomplished ?
  - Wide adoption by Web developer community
  - Not really…
    - Still JavaScript
    - Kind of a big hack

One language to rule them all…

- WebAssembly (March 2017)
  - Best of both worlds: NaCl + Asm.Js
  - W3C standard
  - New stack-based virtual machine
    - Not a sandbox
    - Not JavaScript
  - Uses its own ABI
    - Limited instruction set
    - Strict types
    - **Validate once, run forever**
  - Describes its own file format
    - Simplified version of ELF

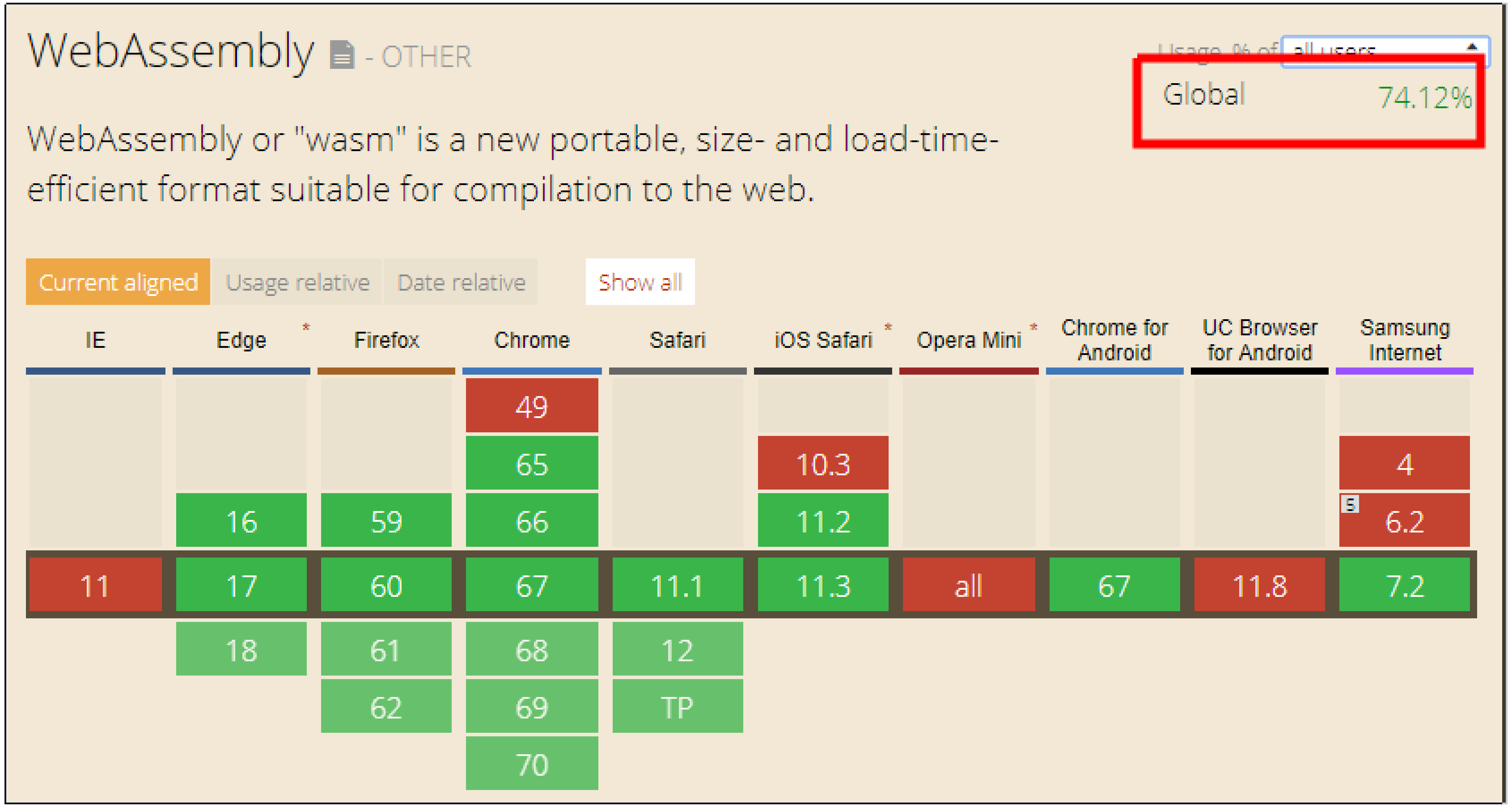
One language to rule them all…

- WebAssembly (March 2017)
  - Close to native performance on simple operations
    - Perfect for client-side computation
  - (Officially) Designed with security in mind
  - MVP fully functional, but easily extensible

- (Rather) Slow adoption
  - Although huge potential
    - 3D games can run smoothly in the browser
    - Real-time application

https://webassembly.org/demo/

One language to rule them all…



WebAssembly - OTHER

Usage % of all users

Global 74.12%

WebAssembly or "wasm" is a new portable, size- and load-time-efficient format suitable for compilation to the web.

IE	Edge *	Firefox	Chrome	Safari	iOS Safari *	Opera Mini *	Chrome for Android	UC Browser for Android	Samsung Internet
			49						
			65		10.3				4
	16	59	66		11.2				6.2
11	17	60	67	11.1	11.3	all	67	11.8	7.2
	18	61	68	12					
		62	69	TP					
			70						

Source: CanIUse.com

In a nutshell

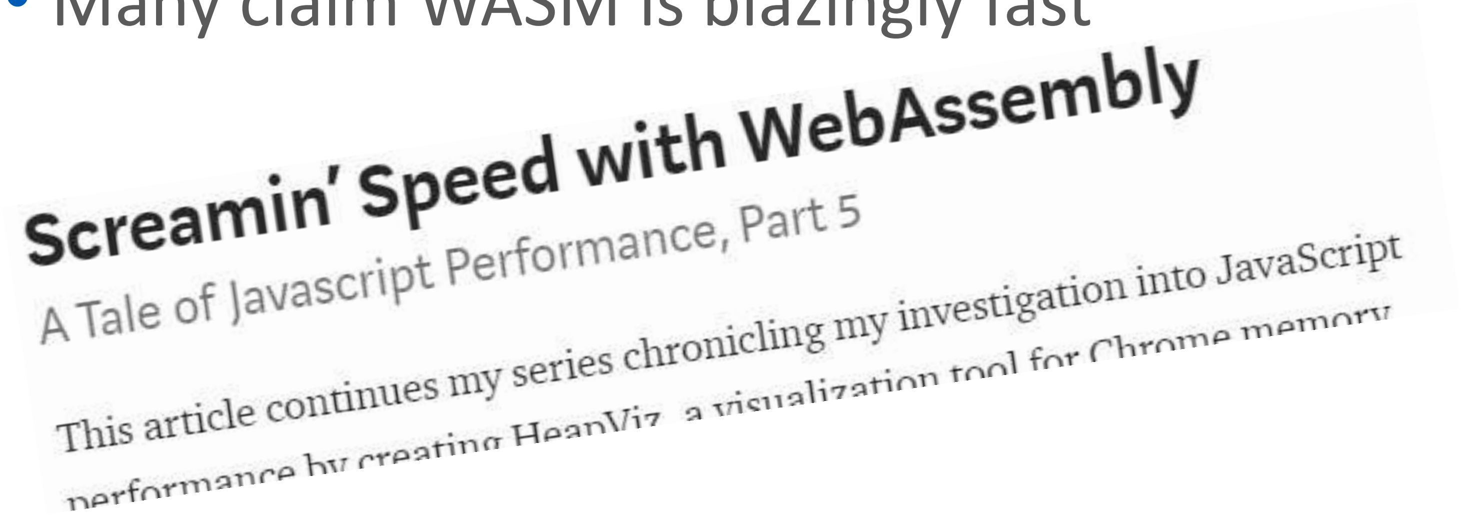
	Native Client (NaCl)	Asm.Js	WebAssembly
Close to native performance	✓	✓	✓
Ahead-of-time compilation	✓	✗	✓
W3C open standard	✗	✓	✓
Security boundaries	✓	✗	✓

In a nutshell

	Native Client (NaCl)	Asm.Js	WebAssembly
Close to native performance	✓	✓	✓
Ahead-of-time compilation			✓
W3C open standard			✓
Security boundaries			✓

Web-Assembly: performance

- Many claim WASM is blazingly fast



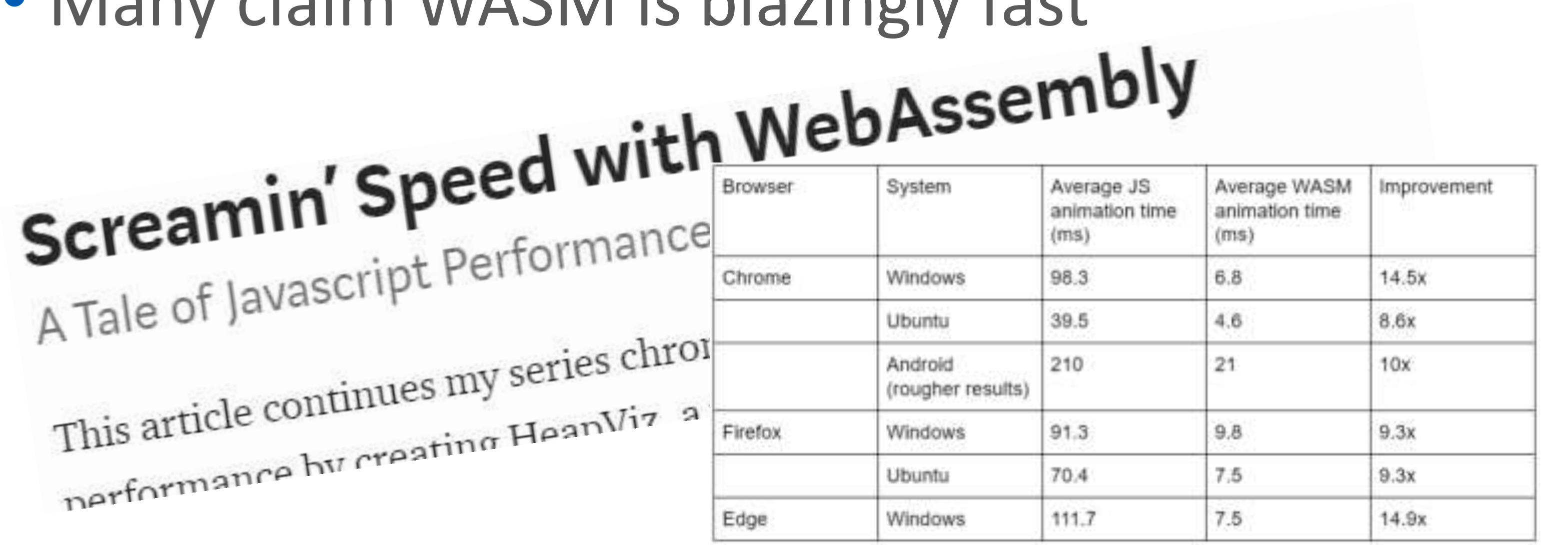
Screamin' Speed with WebAssembly

A Tale of Javascript Performance, Part 5

This article continues my series chronicling my investigation into JavaScript performance by creating HeapViz, a visualization tool for Chrome memory

Web-Assembly: performance

- Many claim WASM is blazingly fast



Screamin' Speed with WebAssembly

A Tale of Javascript Performance

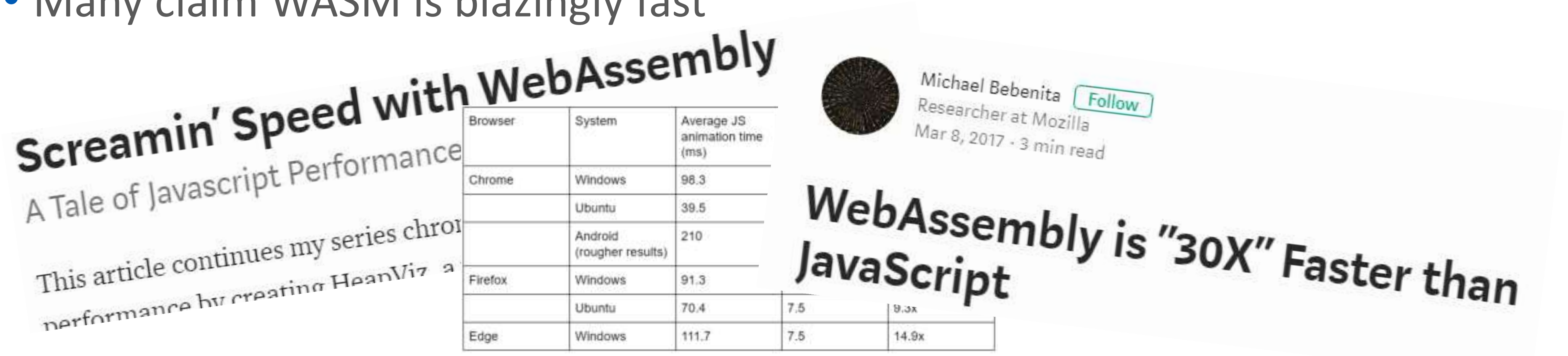
This article continues my series chron

performance by creating HeapViz, a

Browser	System	Average JS animation time (ms)	Average WASM animation time (ms)	Improvement
Chrome	Windows	98.3	6.8	14.5x
	Ubuntu	39.5	4.6	8.6x
	Android (rougher results)	210	21	10x
Firefox	Windows	91.3	9.8	9.3x
	Ubuntu	70.4	7.5	9.3x
Edge	Windows	111.7	7.5	14.9x

Web-Assembly: performance

- Many claim WASM is blazingly fast



Screamin' Speed with WebAssembly

A Tale of Javascript Performance

This article continues my series chro
performance by creating HeapViz, a

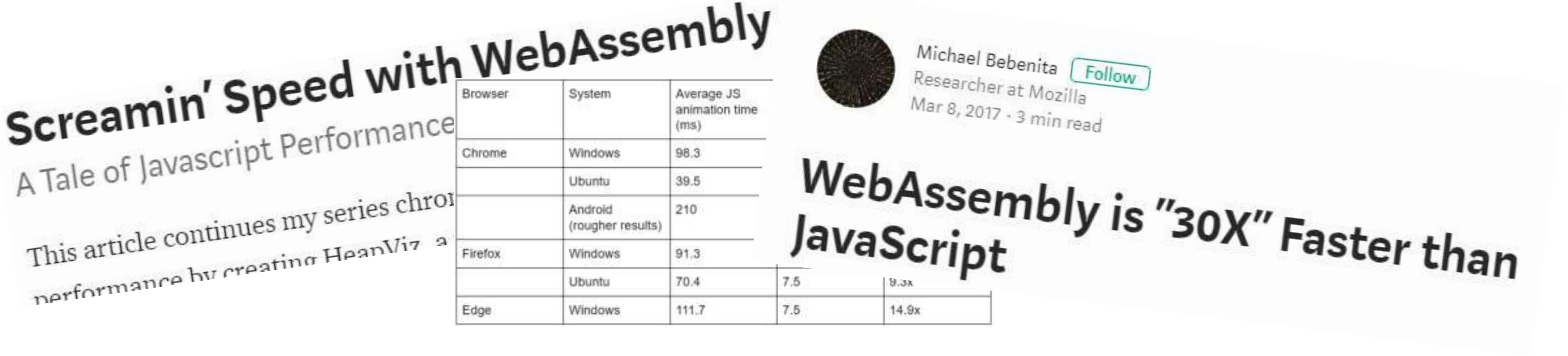
Browser	System	Average JS animation time (ms)		
Chrome	Windows	98.3		
	Ubuntu	39.5		
	Android (rougher results)	210		
Firefox	Windows	91.3		
	Ubuntu	70.4	7.5	9.3x
Edge	Windows	111.7	7.5	14.9x

Michael Bebenita Follow
Researcher at Mozilla
Mar 8, 2017 · 3 min read

WebAssembly is "30X" Faster than JavaScript

Web-Assembly: performance

- Many claim WASM is blazingly fast



Screamin' Speed with WebAssembly

A Tale of Javascript Performance

This article continues my series chron

performance by creating HeapViz a

Browser	System	Average JS animation time (ms)		
Chrome	Windows	98.3		
	Ubuntu	39.5		
	Android (rougher results)	210		
Firefox	Windows	91.3		
	Ubuntu	70.4	7.5	9.3x
Edge	Windows	111.7	7.5	14.9x

Michael Bebenita Follow

Researcher at Mozilla

Mar 8, 2017 · 3 min read

WebAssembly is "30X" Faster than JavaScript

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

WebAssembly: side note

- Not just for the web
  - Run WASM executable
    - https://github.com/AndrewScheidecker/WAVM
  - EOS (WASM for smart contracts)
    - https://twitter.com/ryan_elfmaster/status/996527399138353152
  - MicroKernel that runs WASM
    - https://github.com/nebulet/nebulet
  - WinDbg 10+
    - Embeds ChakraCore from JS scripting support
    - WASM is enabled ☺
    - So yes, you can run WASM code from JS code inside Chakra inside WinDbg!

WebAssembly: side note

- Enough with the marketing pitch, let's dive into WebAssembly !

WebAssembly MVP 1.0

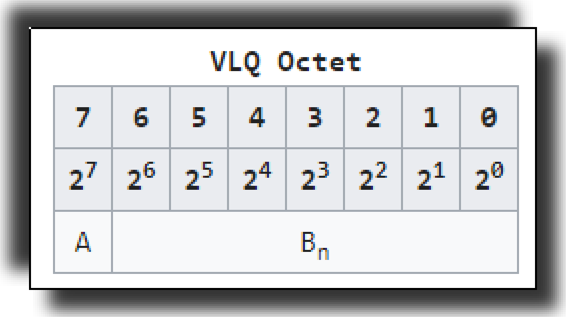
SOPHOS

Web-Assembly Minimum Viable Product (MVP)

- First “stable” release – 1.0 (March 2017)
  - "Viable" = provides a complete runtime environment
  - Leaves room for further extension (“Post-MVP”)

- Strict specification of WASM format
  - Small but (Turing-)complete instruction set (172 instructions)
    - Arithmetic operation (including on float)
    - Load/Store
    - Control-Flow (branch, function call, return, etc.)

  - Stack-based Virtual Machine
    - No registers
    - Little endian
    - Strict types
    - Flat 32-bit address space, page size to 64KB
    - (Web) JIT-Translates WASM bytecode into native code
    - Uses Variable-Length (Unsigned) Integers (varint/varuint) – VLQ
      - 0x7F represented with 1 byte (\x7F)
      - 0x80 represented with 2 bytes (\x81\x00)



VLQ Octet

7	6	5	4	3	2	1	0
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
A	B_n						

WebAssembly Instruction Set (brief) overview

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

Memory access instructions

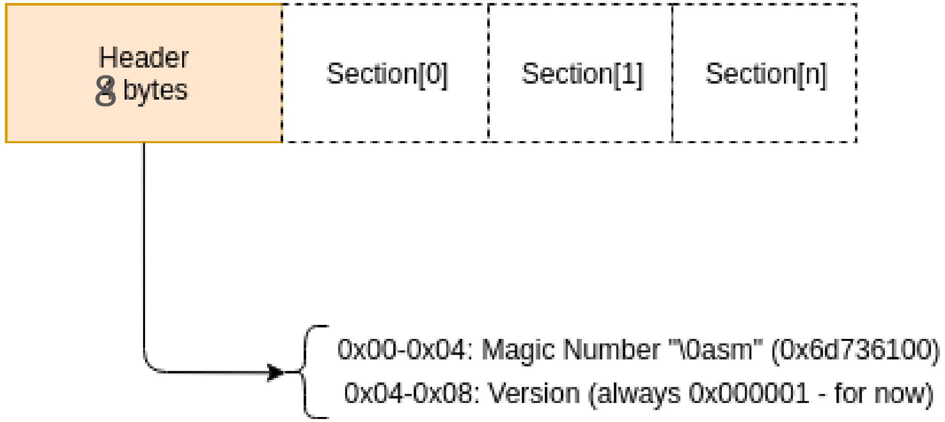
Mnemonic	Opcode	Description
`i32.load / i64.load`	0x28 / 0x29	Load integer from memory
`f32.load / f64.load`	0x2a / 0x2b	Load float from memory
`i32.store / i64.store`	0x36 / 0x37	Store integer from memory
`i32.store / i64.store`	0x36 / 0x37	Store float from memory
`current_memory`	0x3f	Get the current memory size
`grow_memory`	0x40	Increase the memory size

Arithmetic and logic instructions

Mnemonic	Opcode	Description
`i32.add / i32.sub / i32.mul`	0x6a / 0x6b / 0x6c	Add / subtract / divide
`i32.div_s / i32.div_u`	0x6d / 0x6e	(Un)signed Division
`i32.rem_s / i32.rem_u`	0x6f / 0x70	(Un)signed Modulo
`i32.and / i32.xor / i32.or / i32.shl`	0x71 / 0x73 / 0x72 / 0x74	Logic operators

WASM File Format – header

- File format designed for simplicity
  - 8-byte static header
  - (Optionally) N sections



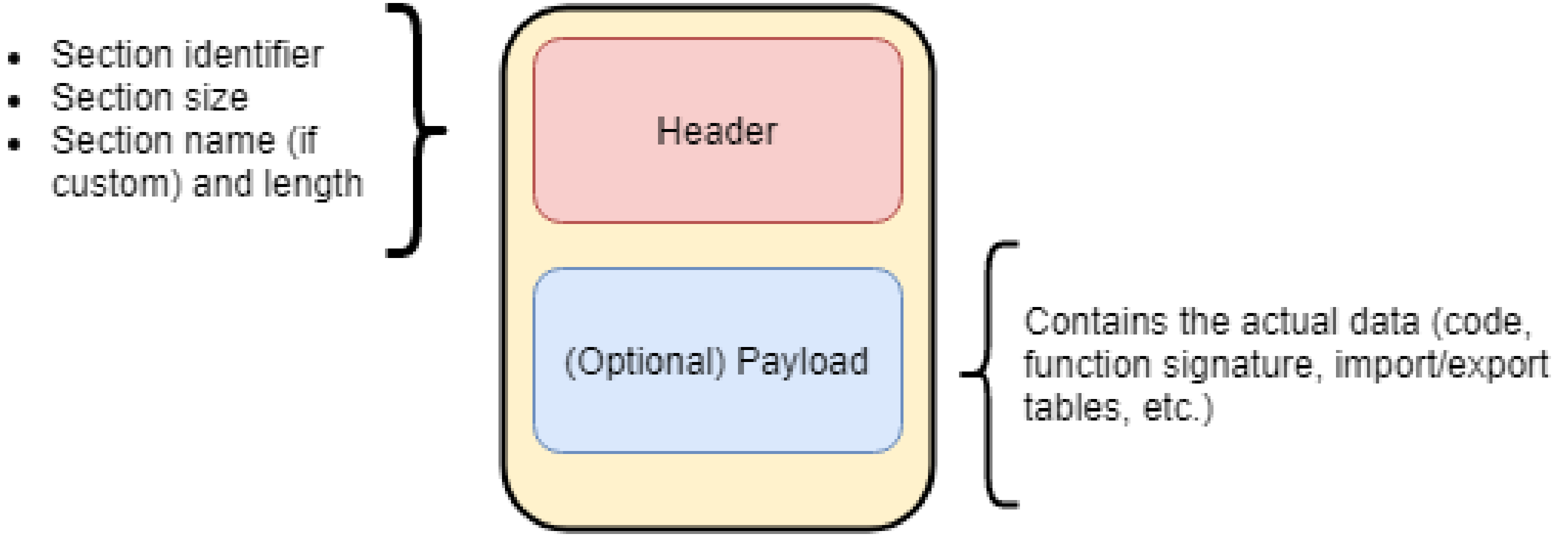
0x00-0x04: Magic Number "\0asm" (0x6d736100)
0x04-0x08: Version (always 0x000001 - for now)

  - The shortest semantically valid WASM file is 8-byte long (header-only)

```
$ printf "\x00\x61\x73\x6d\x01\x00\x00\x00" > ./MyFirstWasmFile.wasm
```

WASM File Format – sections

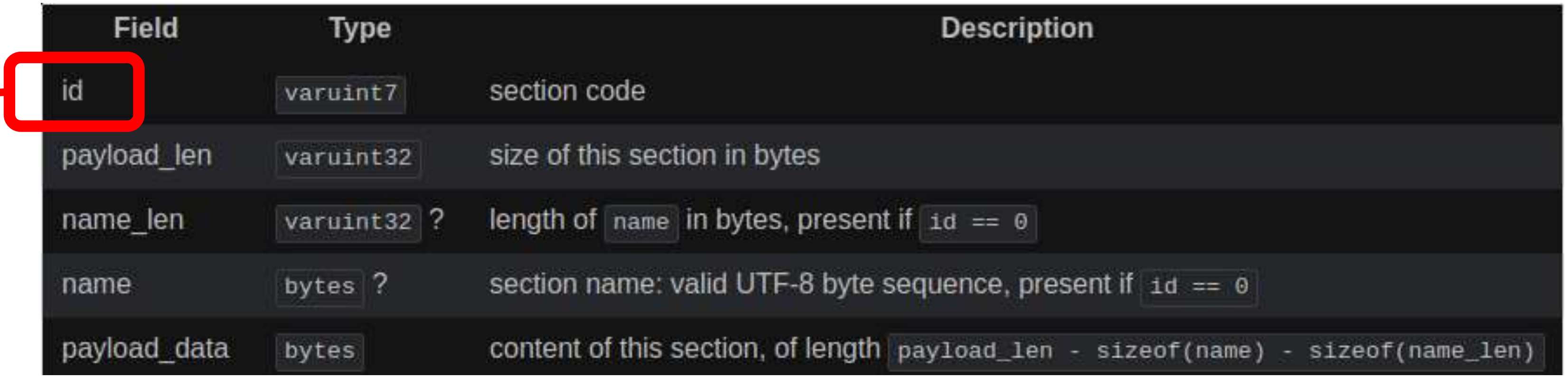
- Each section has a header which states its purpose
- Section sequencing does not matter in the WASM file
    - … But does when parsing it



- Section identifier
- Section size
- Section name (if custom) and length

Contains the actual data (code, function signature, import/export tables, etc.)

WASM File Format – section header



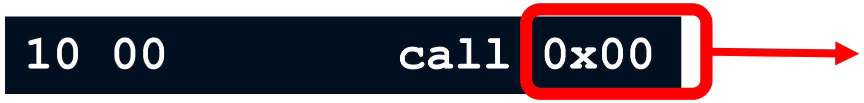
Field	Type	Description
id	`varuint7`	section code
payload_len	`varuint32`	size of this section in bytes
name_len	`varuint32` ?	length of `name` in bytes, present if `id == 0`
name	`bytes` ?	section name: valid UTF-8 byte sequence, present if `id == 0`
payload_data	`bytes`	content of this section, of length `payload_len - sizeof(name) - sizeof(name_len)`

Identifier	Section name	Description
0	Custom	Custom section*
1	Type	Function signature declarations
2	Import	Import declarations
3	Function	Function declarations
4	Table	Indirect function table and other tables
5	Memory	Memory attributes
6	Global	Global declarations
7	Export	Export
8	Start	Start function declaration
9	Element	Elements section
10	Code	Function bodies (code)
11	Data	Data segments

WASM File Format – Sections

- `Function` section
  - Defines the signature of **all** the functions in the current WASM module
  - Including:
    - Number and type of arguments (0 or more)
    - Number and type of return value (at most 1)
  - Stored by index in table

- `Code` section
  - Defines the body (code) of **all** the functions
  - Indexes for entries in `Code` and `Function` section are linked
  - Assembly function calls are specified by this index



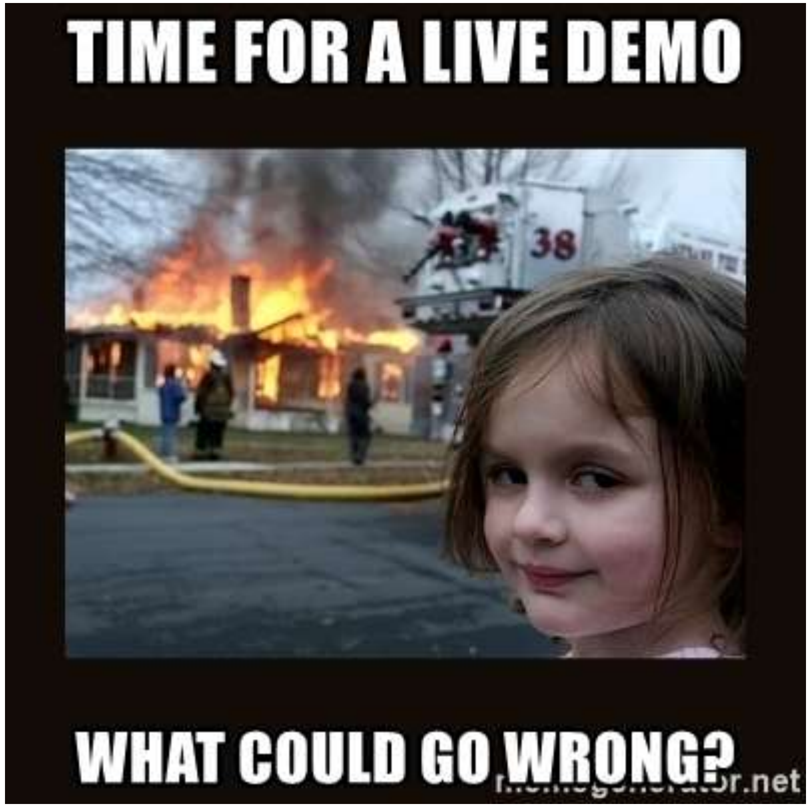
```
10 00          call 0x00
```

Call the function of index **0**

WASM File Format – Sections

- `Export` section - immutable
    - Declares all objects to be exported
    - Allows to expose functions, memory, data
        - Can be consumed by another WASM module, by JavaScript (when running in browser), etc.
- `Import` section - immutable
    - Declares all objects to be imported from another WASM module, from JS, or other
- `Start` section
    - Holds the index to the start function to be called when execution starts (i.e. `main()`-like)
- `Global` section
    - Defines **all** the global variables of the module
    - Can specify whether a variable is mutable or not

Demo


TIME FOR A LIVE DEMO
38
WHAT COULD GO WRONG?
memegenerator.net

WebAssembly: from C to the Web

SOPHOS

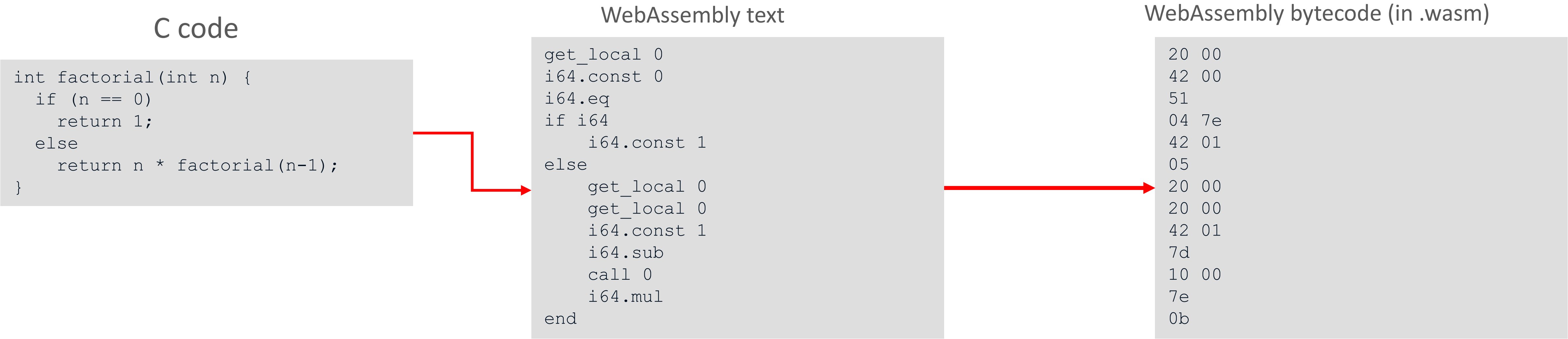
From C to WebAssembly: using tools

- Using toolkits like EmScripten
  - Leverages LLVM IR to convert C code to WASM code
    - Trivial to create WASM code using `emcc` compiler
  - For Web apps, also generates a JavaScript loader + HTML (ugly) scaffold

```
> rustc --target=wasm32-unknown-emscripten hello.rs -o hello.html
```

```
> GOOS=js GOARCH=wasm go build hello.go
```

```
> emcc -s WASM=1 factorial.c
```



C code

```
int factorial(int n) {
  if (n == 0)
    return 1;
  else
    return n * factorial(n-1);
}
```

WebAssembly text

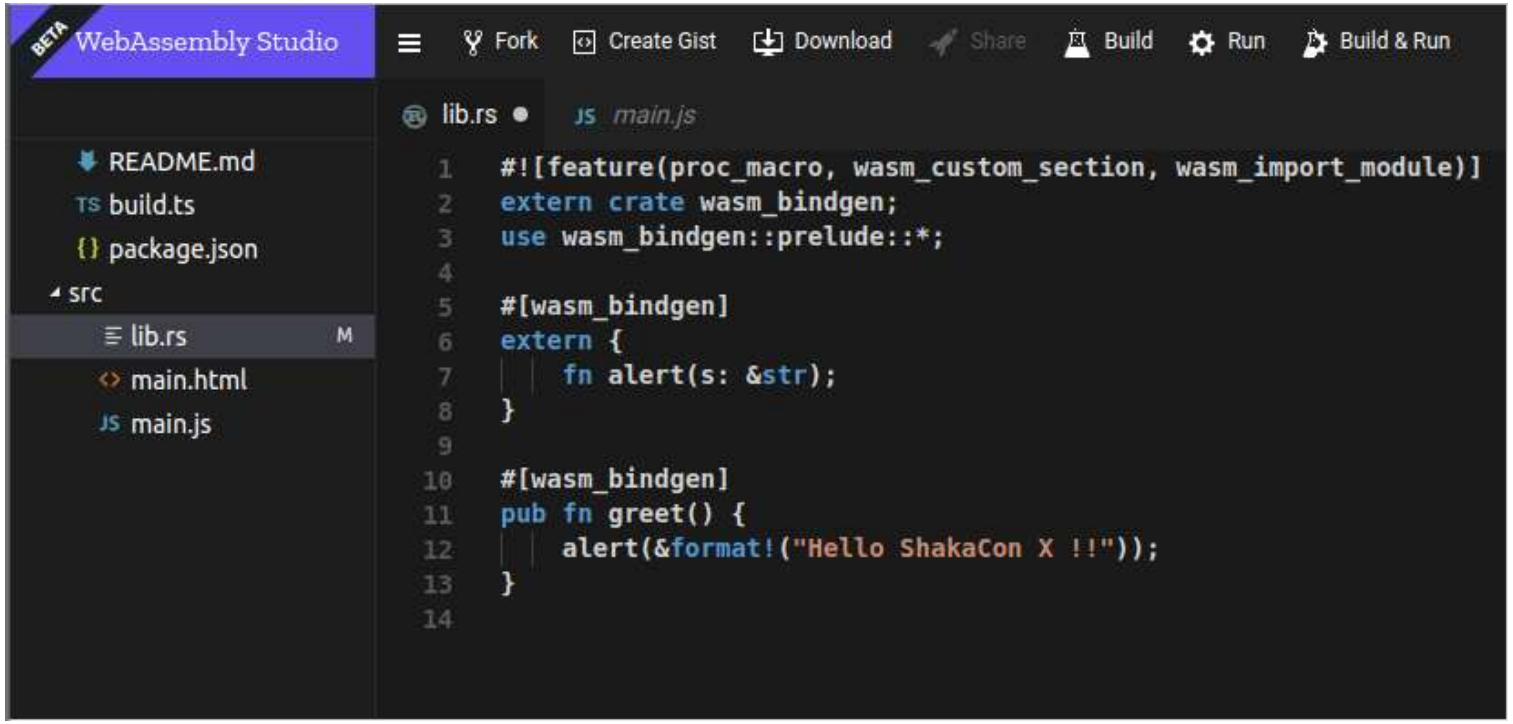
```
get_local 0
i64.const 0
i64.eq
if i64
    i64.const 1
else
    get_local 0
    get_local 0
    i64.const 1
    i64.sub
    call 0
    i64.mul
end
```

WebAssembly bytecode (in .wasm)

```
20 00
42 00
51
04 7e
42 01
05
20 00
20 00
42 01
7d
10 00
7e
0b
```

From C to WebAssembly: using tools

- WASM Studio
  - Web IDE that sets up an environment to write C or Rust
  - Outputs .WASM + JS + HTML

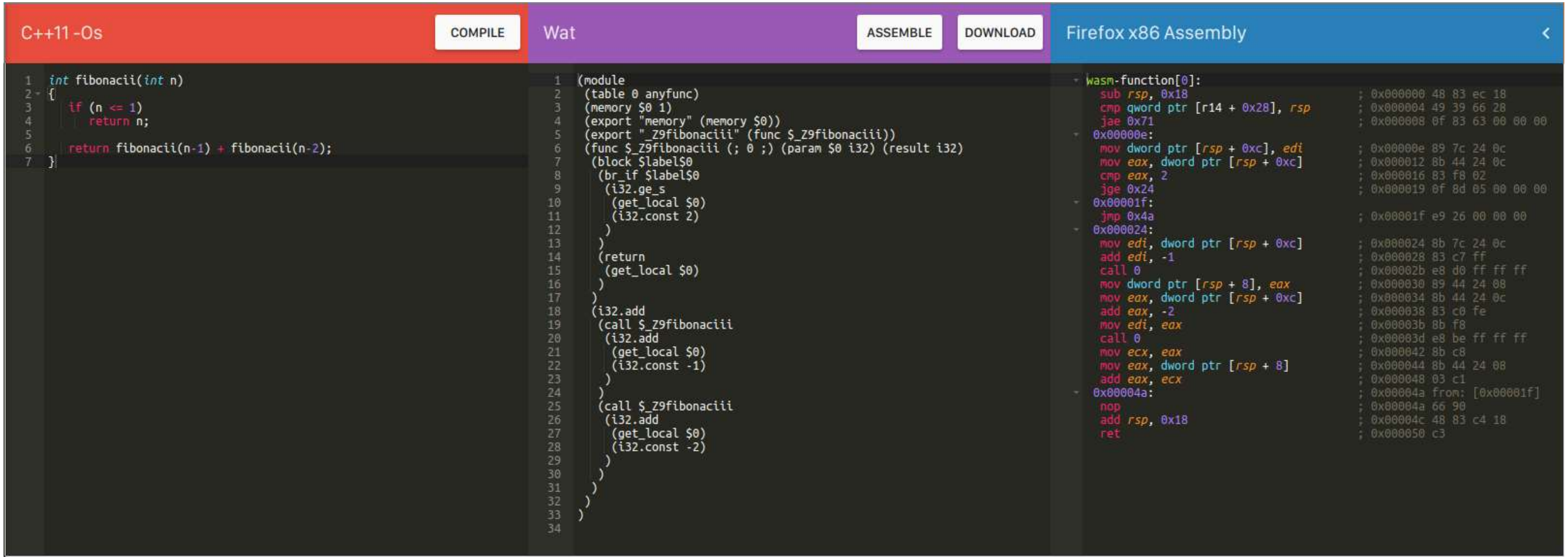


```rust
#![feature(proc_macro, wasm_custom_section, wasm_import_module)]
extern crate wasm_bindgen;
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
extern {
    fn alert(s: &str);
}

#[wasm_bindgen]
pub fn greet() {
    alert(&format!("Hello ShakaCon X !!"));
}

```

https://webassembly.studio/

From C to WebAssembly

- WASM Explorer helps understands the transformation done
  - Similar to Godbolt Compiler Explorer (but for WASM)



```
int fibonacii(int n)
{
   if (n <= 1)
      return n;

   return fibonacii(n-1) + fibonacii(n-2);
}
```

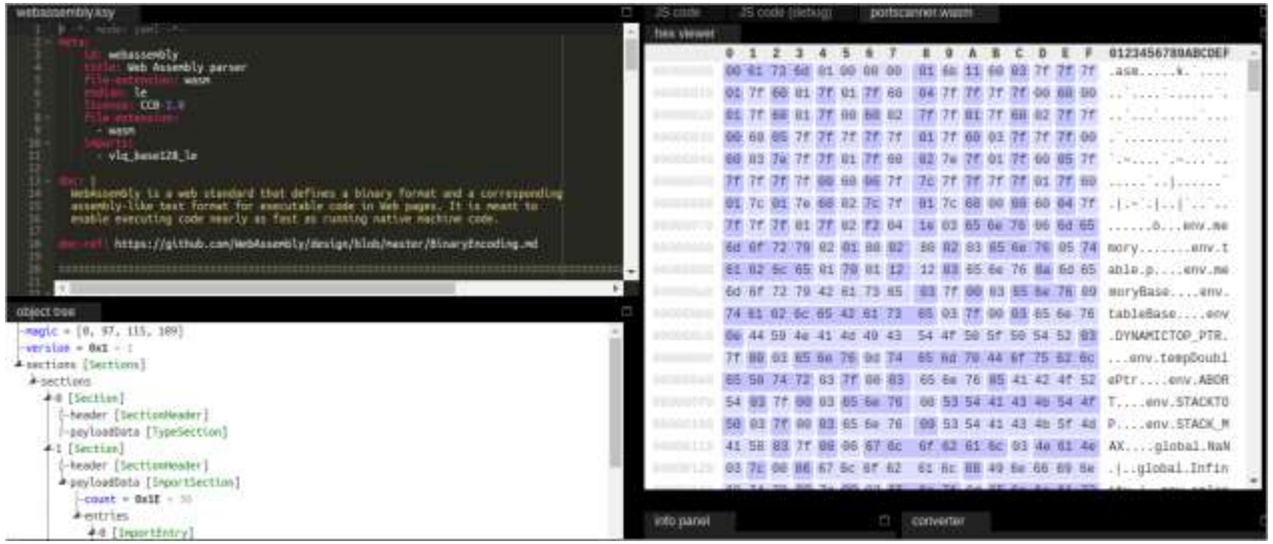
```
(module
 (table 0 anyfunc)
 (memory $0 1)
 (export "memory" (memory $0))
 (export "_Z9fibonacii" (func $_Z9fibonacii))
 (func $_Z9fibonacii (; 0 ;) (param $0 i32) (result i32)
  (block $label$0
   (br_if $label$0
    (i32.ge_s
     (get_local $0)
     (i32.const 2)
    )
   )
   (return
    (get_local $0)
   )
  )
  (i32.add
   (call $_Z9fibonacii
    (i32.add
     (get_local $0)
     (i32.const -1)
    )
   )
   (call $_Z9fibonacii
    (i32.add
     (get_local $0)
     (i32.const -2)
    )
   )
  )
 )
)
```

```
wasm-function[0]:
  sub rsp, 0x18                          ; 0x000000 48 83 ec 18
  cmp qword ptr [r14 + 0x28], rsp        ; 0x000004 49 39 66 28
  jae 0x71                               ; 0x000008 0f 83 63 00 00 00
 0x00000e:
  mov dword ptr [rsp + 0xc], edi         ; 0x00000e 89 7c 24 0c
  mov eax, dword ptr [rsp + 0xc]         ; 0x000012 8b 44 24 0c
  cmp eax, 2                             ; 0x000016 83 f8 02
  jge 0x24                               ; 0x000019 0f 8d 05 00 00 00
 0x00001f:
  jmp 0x4a                               ; 0x00001f e9 26 00 00 00
 0x000024:
  mov edi, dword ptr [rsp + 0xc]         ; 0x000024 8b 7c 24 0c
  add edi, -1                            ; 0x000028 83 c7 ff
  call 0                                 ; 0x00002b e8 d0 ff ff ff
  mov dword ptr [rsp + 8], eax           ; 0x000030 89 44 24 08
  mov eax, dword ptr [rsp + 0xc]         ; 0x000034 8b 44 24 0c
  add eax, -2                            ; 0x000038 83 c0 fe
  mov edi, eax                           ; 0x00003b 8b f8
  call 0                                 ; 0x00003d e8 be ff ff ff
  mov ecx, eax                           ; 0x000042 8b c8
  mov eax, dword ptr [rsp + 8]           ; 0x000044 8b 44 24 08
  add eax, ecx                           ; 0x000048 03 c1
 0x00004a:                               ; 0x00004a from: [0x00001f]
  nop                                    ; 0x00004a 66 90
  add rsp, 0x18                          ; 0x00004c 48 83 c4 18
  ret                                    ; 0x000050 c3
```

https://mbebenita.github.io/WasmExplorer/

Our custom toolkit

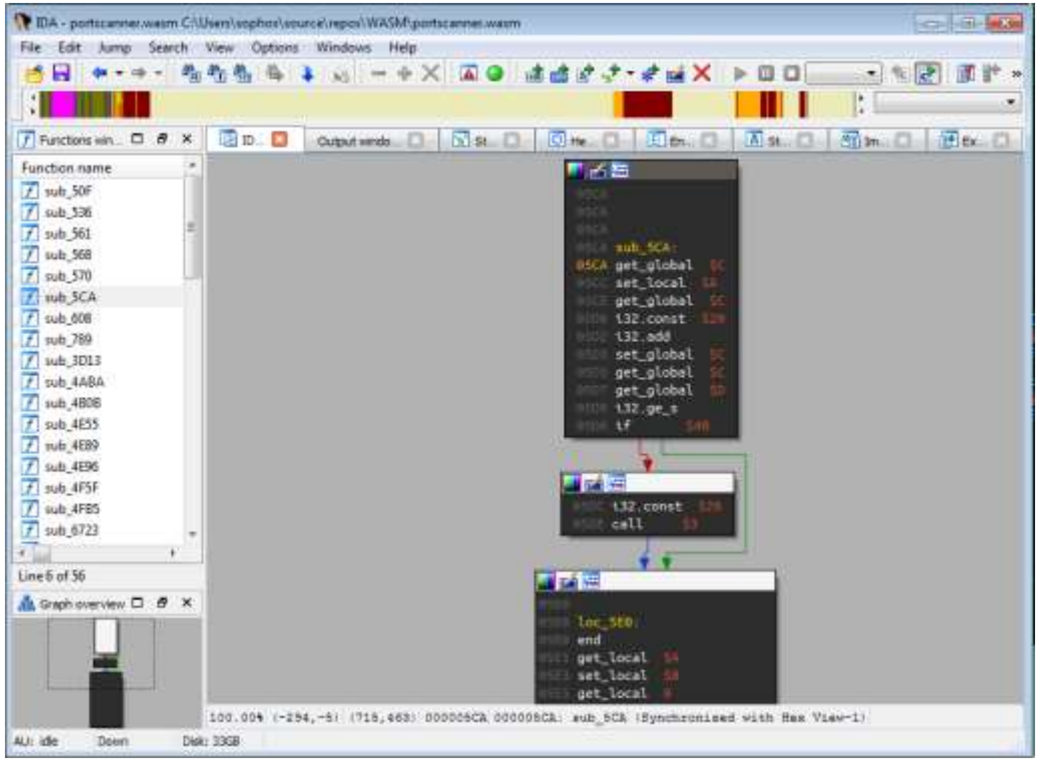
- Kaitai.io parser
- Kaitai-struct-based disassembler



```
# -*- 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
00 61 73 6d 01 00 00 00  01 6e 11 60 03 7f 7f 7f  .asm.....k......
01 7f 60 01 7f 01 7f 60  04 7f 7f 7f 7f 00 60 00  ................
01 7f 60 01 7f 00 60 02  7f 7f 01 7f 60 02 7f 7f  ................
00 60 05 7f 7f 7f 7f 7f  01 7f 60 03 7f 7f 7f 00  ................
60 03 7e 7f 7f 01 7f 60  02 7e 7f 01 7f 60 05 7f  `.~....`.~...`..
7f 7f 7f 7f 00 60 06 7f  7c 7f 7f 7f 7f 01 7f 60  .....`..|......`
01 7c 01 7e 60 02 7c 7f  01 7c 60 00 00 60 04 7f  .|.~`.|..|`..`..
7f 7f 7f 01 7f 02 f2 04  1e 03 65 6e 76 06 6d 65  .......o...env.me
6d 6f 72 79 02 01 80 02  80 02 03 65 6e 76 05 74  mory.......env.t
61 62 6c 65 01 70 01 12  12 03 65 6e 76 0a 6d 65  able.p....env.me
6d 6f 72 79 42 61 73 65  03 7f 00 03 65 6e 76 09  moryBase....env.
74 61 62 6c 65 42 61 73  65 03 7f 00 03 65 6e 76  tableBase....env
0e 44 59 4e 41 4d 49 43  54 4f 50 5f 50 54 52 03  .DYNAMICTOP_PTR.
7f 00 03 65 6e 76 0d 74  65 6d 70 44 6f 75 62 6c  ...env.tempDoubl
65 50 74 72 03 7f 00 03  65 6e 76 05 41 42 4f 52  ePtr....env.ABOR
54 03 7f 00 03 65 6e 76  0b 53 54 41 43 4b 54 4f  T....env.STACKTO
50 03 7f 00 03 65 6e 76  0d 53 54 41 43 4b 5f 4d  P....env.STACK_M
41 58 03 7f 00 06 67 6c  6f 62 61 6c 03 4e 61 4e  AX....global.NaN
03 7c 00 06 67 6c 6f 62  61 6c 08 49 6e 66 69 6e  .|..global.Infin
```



sub_50F
sub_536
sub_561
sub_568
sub_570
sub_5CA
sub_608
sub_789
sub_3D13
sub_4ABA
sub_4B08
sub_4E55
sub_4E89
sub_4E96
sub_4F5F
sub_4FB5
sub_6723

Line 6 of 56

```
sub_5CA:
05CA get_global
     set_local
     get_global
     i32.const
     i32.add
     set_global
     get_global
     get_global
     i32.ge_s
     if

     i32.const
     call

loc_5E0:
     end
     get_local
     set_local
     get_local
```

100.00% (-294,-5) (718,463) 000005CA 000005CA: sub_5CA (Synchronised with Hex View-1)

AU: idle Down Disk: 33GB

- IDA Pro loader & processor
- All tools to be released on GitHub after the talk

Into the Web Browser

- Compilers (like EmScripten) will generate a valid WASM file (with JS + HTML loaders)
- `.wasm` is not a native MIME type for Web Browser, we need a loader
- Let’s use JavaScript!
  - Fetch the WASM bytecode into a JS byte array (2 ways)

    1 - If the WASM module is small enough (< 4096 bytes),

    it can be inlined directly inside the JS code.

```
<html>
 <body>
  <script>
    var WasmArrayBuffer = new Uint8Array(SIZE);
    RawWasm[0] = 0x00;                       ‘\0’
    RawWasm[1] = 0x61;                       ‘a’
    RawWasm[2] = 0x73;                       ‘s’
    RawWasm[3] = 0x6d;                       ‘m’
    […]
```

Into the Web Browser

- (Cont.) Read the WASM bytecode from .wasm file (2 ways)

  2 – Using EcmaScript 6 Promise feature (recommended approach)

```
<html>
 <body>
  <script>
    document.addEventListener("DOMContentLoaded", StartWasm);

    function StartWasm(event) {
            fetch('hello-world.wasm').then(response =>
                       response.arrayBuffer()
            );
            […]
```

- The JavaScript `ArrayBuffer` containing the WASM module is validated syntaxically:

```
var MyModule = new WebAssembly.Module(WasmArrayBuffer);          // strict format, signature, types checks are done there
```

- The JS engine then parses the code, and JITs-it and issues native binary code.

Into the Web Browser

- (Cont.) Declare a N-page large memory area and the JavaScript exports

```
function bar(){}

var memory = new WebAssembly.Memory({ initial : 20 });  // size=20*65536B

const exported = {
    stdlib: { foo: bar },                 // exposes “stdlib.foo” to WASM code
                                          // but execute JS function bar

    js: { memory: memory }                // pass the reference to the memory area to VM
};
```

- Finally, the VM can be instantiated

```
var MyInstance = new WebAssembly.Instance(MyModule, exported);
```

- The VM is running, can be interacted with via exported functions

```
MyInstance.exports.ComputePi();
```

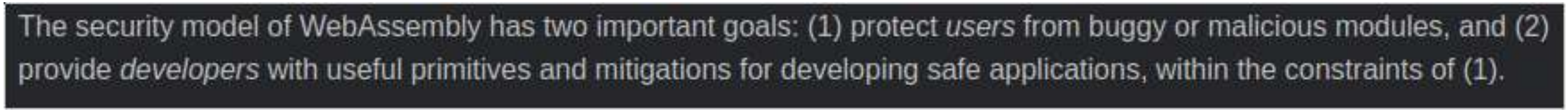
Demo

Web-(Dis)Assembly:
Attack Surface Analysis

SOPHOS

WASM Security Consideration: the theory

- Entire section in the specification dedicated to security



The security model of WebAssembly has two important goals: (1) protect *users* from buggy or malicious modules, and (2) provide *developers* with useful primitives and mitigations for developing safe applications, within the constraints of (1).

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

- ○ (1) Sandboxed code + Same-Origin-Policy (SOP) enforced
  - Strict isolation
  - enforcement of SOP (including CSP and HSTS)

- ○ (2) Immutable code + Control-Flow Integrity (CFI) + Separated stack (for return address) + static types
  - Prevent buffer overflow exploitation
  - Code cannot be written for execution **after** the module is parsed
  - Cannot control code pointers, no traditional ROP/JOP
  - No type confusion

WASM Security Consideration: the theory

- Entire section in the specification dedicated to security

> The security model of WebAsse from buggy or malicious modules, and (2)
> provide *developers* with useful p plications, within the constraints of (1).

  - [https://github.com/V ity.md](https://github.com/V)

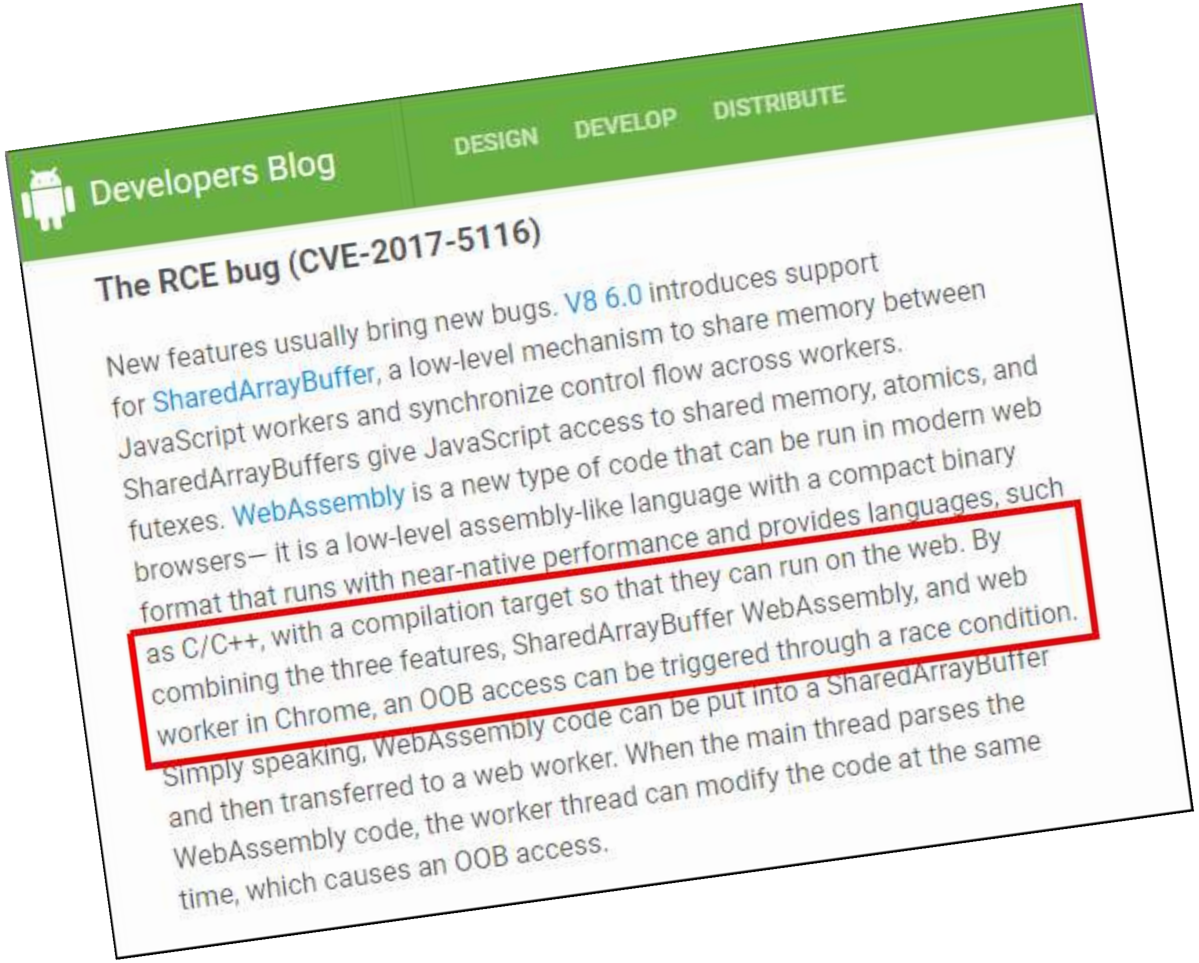
- ○ (1) Sandboxed code +
  - Strict isolation, enfo n loading WASM module
- ○ (2) Immutable code + ated stack (for return address) + static types
  - Prevent buffer overfl
  - Code cannot be writt ed
  - Cannot control code
  - No type confusion

WASM Security Consideration: the reality

- Attack vectors against the protocol/specification
  - Pretty hard…
    - Function return hijacking: similar to traditional ROP, but by forcing return to existing indexes
    - Race Conditions (TOCTOU): no atomicity of operation is guaranteed, memory can be shared
    - Time-based Side-Channel attack (Spectre-like)

- Attack vectors against the implementations
  - Easier (but not easy)…
    - Most implementations are done in C/C++*
    - Back to “traditional” vulnerabilities
      - Memory corruption
      - Race condition

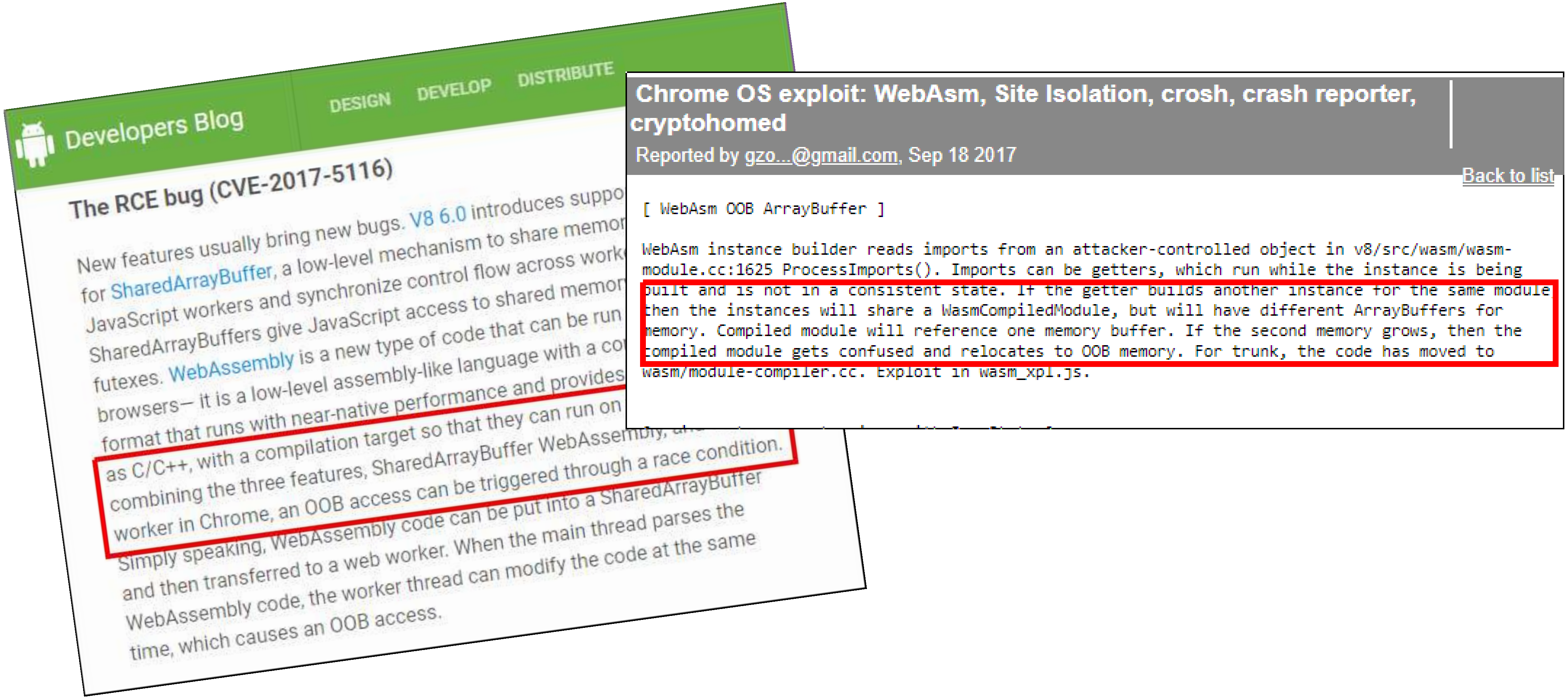
WASM Security Consideration: the reality



The RCE bug (CVE-2017-5116)

New features usually bring new bugs. V8 6.0 introduces support for SharedArrayBuffer, a low-level mechanism to share memory between JavaScript workers and synchronize control flow across workers. SharedArrayBuffers give JavaScript access to shared memory, atomics, and futexes. WebAssembly is a new type of code that can be run in modern web browsers— it is a low-level assembly-like language with a compact binary format that runs with near-native performance and provides languages, such as C/C++, with a compilation target so that they can run on the web. By combining the three features, SharedArrayBuffer WebAssembly, and web worker in Chrome, an OOB access can be triggered through a race condition. Simply speaking, WebAssembly code can be put into a SharedArrayBuffer and then transferred to a web worker. When the main thread parses the WebAssembly code, the worker thread can modify the code at the same time, which causes an OOB access.

WASM Security Consideration: the reality



The RCE bug (CVE-2017-5116)

New features usually bring new bugs. V8 6.0 introduces suppo
for SharedArrayBuffer, a low-level mechanism to share memor
JavaScript workers and synchronize control flow across work
SharedArrayBuffers give JavaScript access to shared memor
futexes. WebAssembly is a new type of code that can be run
browsers— it is a low-level assembly-like language with a co
format that runs with near-native performance and provides
as C/C++, with a compilation target so that they can run on
combining the three features, SharedArrayBuffer WebAssembly, an
worker in Chrome, an OOB access can be triggered through a race condition.
Simply speaking, WebAssembly code can be put into a SharedArrayBuffer
and then transferred to a web worker. When the main thread parses the
WebAssembly code, the worker thread can modify the code at the same
time, which causes an OOB access.

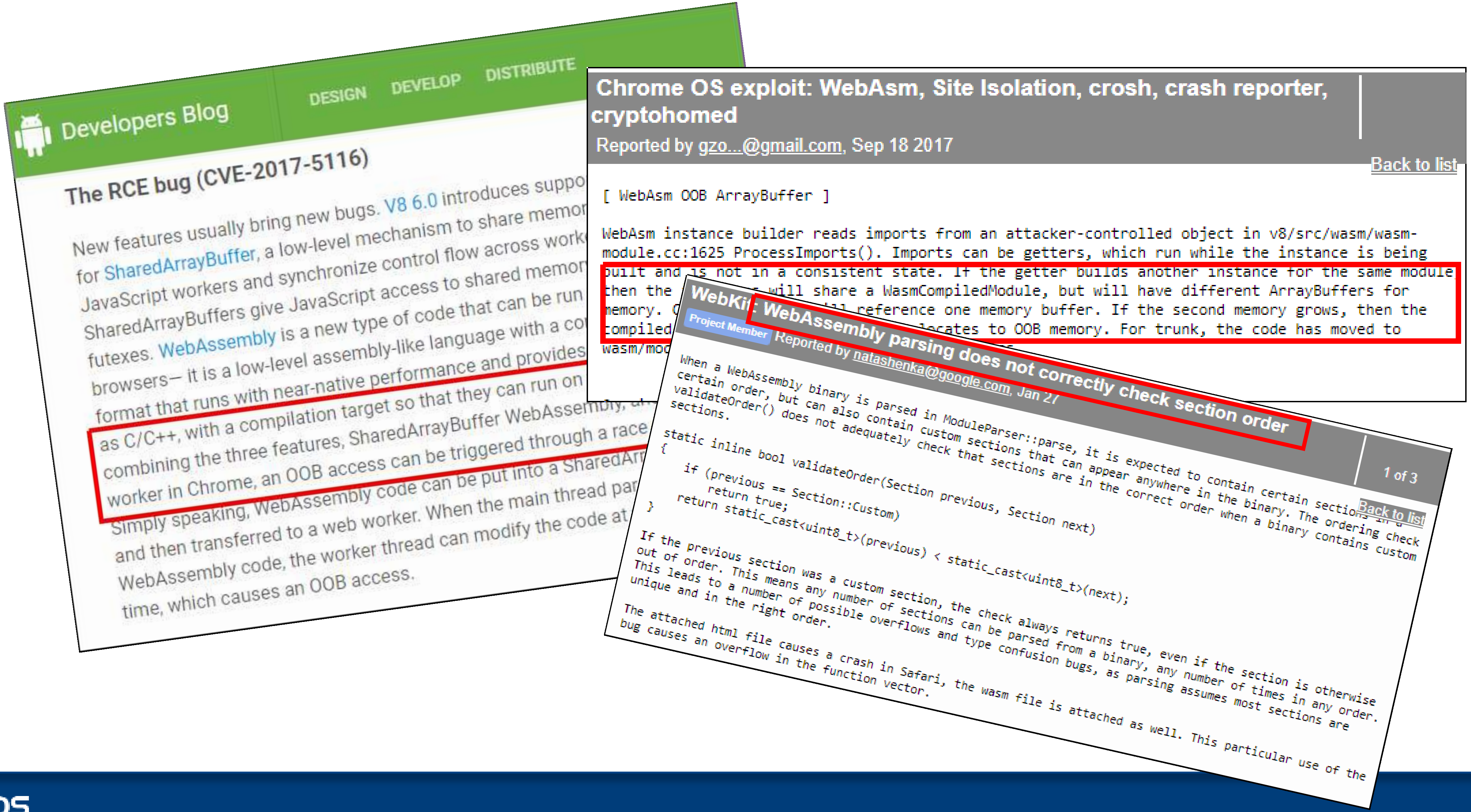
Chrome OS exploit: WebAsm, Site Isolation, crosh, crash reporter, cryptohomed

Reported by gzo...@gmail.com, Sep 18 2017

Back to list

```
[ WebAsm OOB ArrayBuffer ]

WebAsm instance builder reads imports from an attacker-controlled object in v8/src/wasm/wasm-
module.cc:1625 ProcessImports(). Imports can be getters, which run while the instance is being
built and is not in a consistent state. If the getter builds another instance for the same module
then the instances will share a WasmCompiledModule, but will have different ArrayBuffers for
memory. Compiled module will reference one memory buffer. If the second memory grows, then the
compiled module gets confused and relocates to OOB memory. For trunk, the code has moved to
wasm/module-compiler.cc. Exploit in wasm_xpl.js.
```

WASM Security Consideration: the reality



The RCE bug (CVE-2017-5116)

New features usually bring new bugs. V8 6.0 introduces suppo
for SharedArrayBuffer, a low-level mechanism to share memor
JavaScript workers and synchronize control flow across work
SharedArrayBuffers give JavaScript access to shared memor
futexes. WebAssembly is a new type of code that can be run
browsers— it is a low-level assembly-like language with a co
format that runs with near-native performance and provides
as C/C++, with a compilation target so that they can run on
combining the three features, SharedArrayBuffer WebAssembly, a
worker in Chrome, an OOB access can be triggered through a race
Simply speaking, WebAssembly code can be put into a SharedArr
and then transferred to a web worker. When the main thread par
WebAssembly code, the worker thread can modify the code at
time, which causes an OOB access.

Chrome OS exploit: WebAsm, Site Isolation, crosh, crash reporter, cryptohomed

Reported by gzo...@gmail.com, Sep 18 2017

Back to list

```
[ WebAsm OOB ArrayBuffer ]

WebAsm instance builder reads imports from an attacker-controlled object in v8/src/wasm/wasm-
module.cc:1625 ProcessImports(). Imports can be getters, which run while the instance is being
built and is not in a consistent state. If the getter builds another instance for the same module
then the          will share a WasmCompiledModule, but will have different ArrayBuffers for
memory. C          ll reference one memory buffer. If the second memory grows, then the
compiled                 locates to OOB memory. For trunk, the code has moved to
wasm/moc
```

WebKit: WebAssembly parsing does not correctly check section order

Project Member Reported by natashenka@google.com, Jan 27

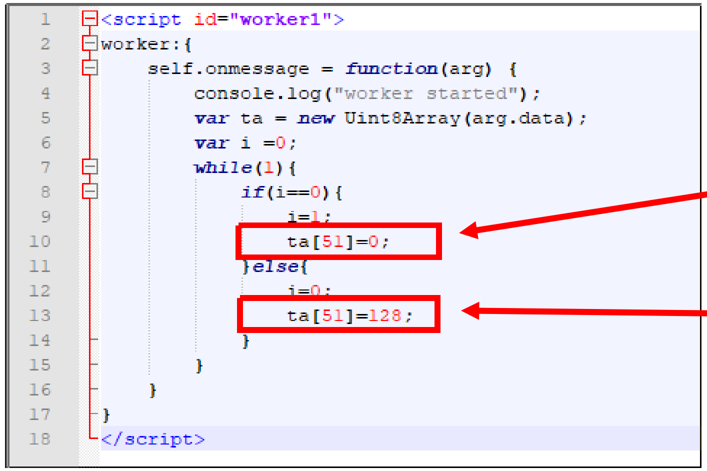
1 of 3

Back to list

```
When a WebAssembly binary is parsed in ModuleParser::parse, it is expected to contain certain sections in a
certain order, but can also contain custom sections that can appear anywhere in the binary. The ordering check
validateOrder() does not adequately check that sections are in the correct order when a binary contains custom
sections.

static inline bool validateOrder(Section previous, Section next)
{
    if (previous == Section::Custom)
        return true;
    return static_cast<uint8_t>(previous) < static_cast<uint8_t>(next);
}

If the previous section was a custom section, the check always returns true, even if the section is otherwise
out of order. This means any number of sections can be parsed from a binary, any number of times in any order.
This leads to a number of possible overflows and type confusion bugs, as parsing assumes most sections are
unique and in the right order.

The attached html file causes a crash in Safari, the wasm file is attached as well. This particular use of the
bug causes an overflow in the function vector.
```

WebAssembly: the interesting case of CVE-2017-5116

- Discovered and used by Qihoo 360 in Google Pixel exploit chain
  - The Memory section passed to the WASM engine is backed by a `SharedArrayBuffer`
  - Attacker can create a worker that will try to change the index of a function during a `call` instruction
  - Race condition (TOCTOU): by winning, the attacker can redirect execution to a controlled index, bypassing the WASM parser validation!



```
<script id="worker1">
worker:{
    self.onmessage = function(arg) {
        console.log("worker started");
        var ta = new Uint8Array(arg.data);
        var i =0;
        while(1){
            if(i==0){
                i=1;
                ta[51]=0;
            }else{
                i=0;
                ta[51]=128;
            }
        }
    }
}
</script>
```

The instruction is checked as `call 0` (valid index)

But executed as `call 128` (out-of-bound index)

WebAssembly: the interesting case of CVE-2017-5116

- Very powerful vulnerability…
  - Once validated, the code is never checked again.

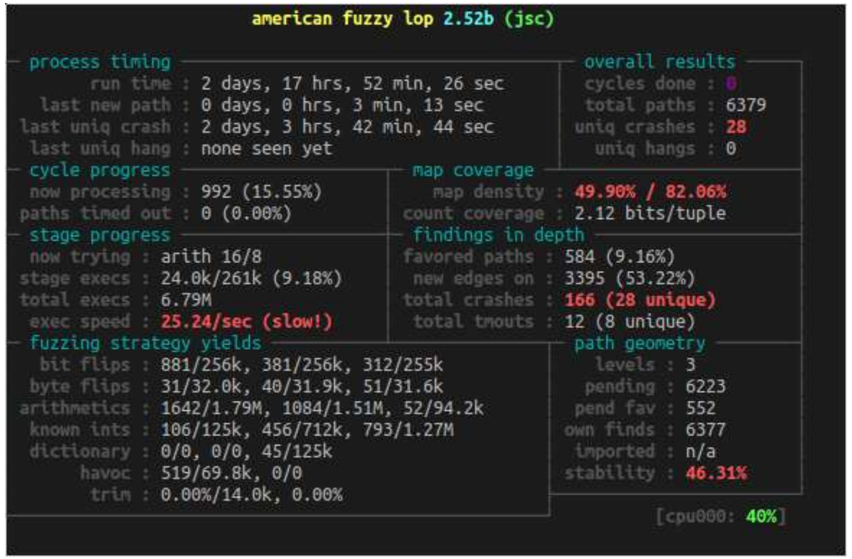
- … But unlikely to be seen in the future
  - `SharedArrayBuffer` objects are now removed / disabled on all recent versions of Web browsers
    - Thank you Spectre !
      - Ironically, Spectre vulnerability would have been a perfect candidate for a WASM module:)

WASM & Web Browsers

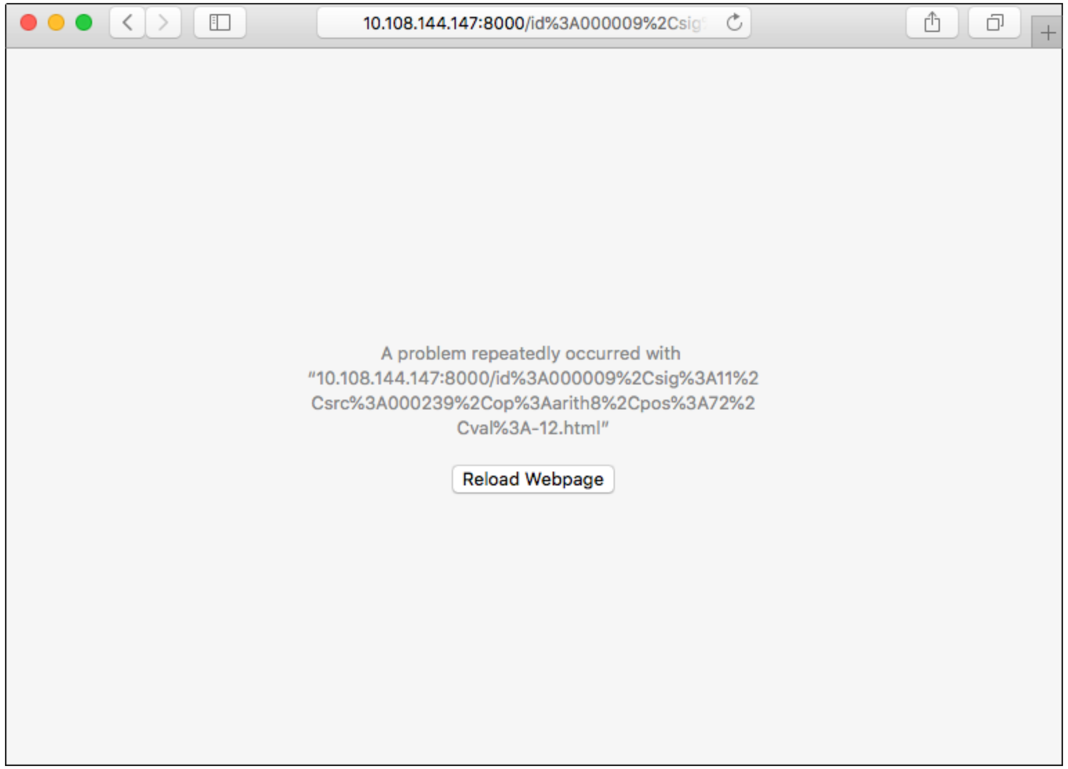
- We’ve reviewed the implementations of the major Web browsers
- All JavaScript engine implementations are Open-Source!
  - ChakraCore (Edge)
    - ~ 48K LoC
  - SpiderMonkey (Firefox)
    - ~47K LoC
  - V8 (Chrome)
    - ~28K LoC
  - JSC (WebKit)
    - ~15K LoC

WASM & Web Browsers

- After ~2 weeks of fuzzing with different tools and strategies, JSC (WebKit) showed a glimpse of hope
  - Specifically crafted WASM file would make `com.apple.WebKit.WebContent` crash



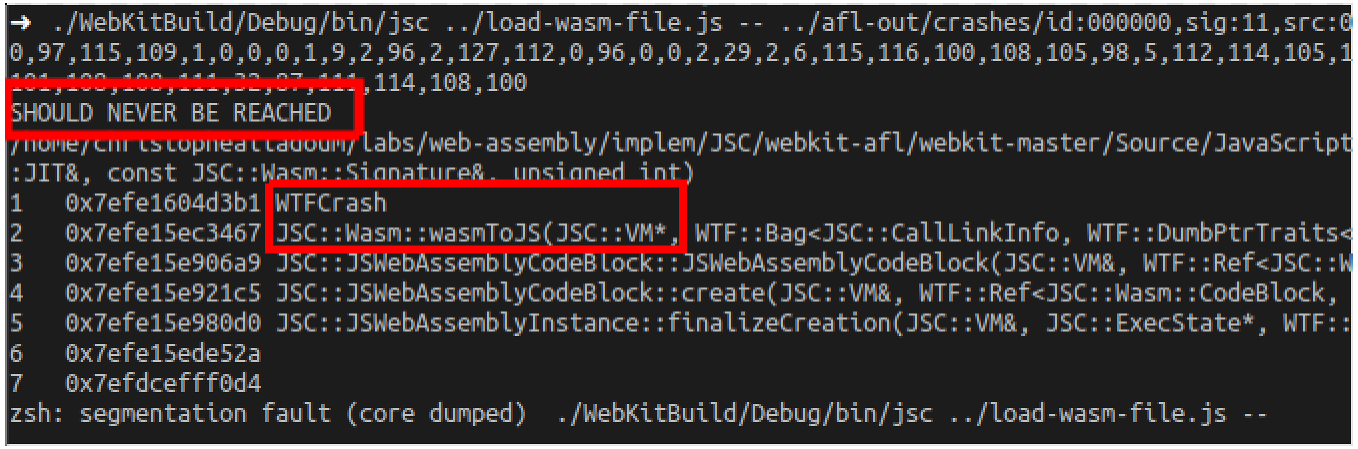
```
                    american fuzzy lop 2.52b (jsc)

- process timing ---------------------------------- overall results ---
        run time : 2 days, 17 hrs, 52 min, 26 sec  | cycles done : 0
   last new path : 0 days, 0 hrs, 3 min, 13 sec    | total paths : 6379
 last uniq crash : 2 days, 3 hrs, 42 min, 44 sec   | uniq crashes : 28
  last uniq hang : none seen yet                   |   uniq hangs : 0
- cycle progress ------------------- map coverage ---------------------
  now processing : 992 (15.55%)    |    map density : 49.90% / 82.06%
 paths timed out : 0 (0.00%)       | count coverage : 2.12 bits/tuple
- stage progress ------------------- findings in depth ----------------
  now trying : arith 16/8          | favored paths : 584 (9.16%)
 stage execs : 24.0k/261k (9.18%)  |  new edges on : 3395 (53.22%)
 total execs : 6.79M               | total crashes : 166 (28 unique)
  exec speed : 25.24/sec (slow!)   |  total tmouts : 12 (8 unique)
- fuzzing strategy yields -------------------------- path geometry ----
   bit flips : 881/256k, 381/256k, 312/255k       |    levels : 3
  byte flips : 31/32.0k, 40/31.9k, 51/31.6k       |   pending : 6223
 arithmetics : 1642/1.79M, 1084/1.51M, 52/94.2k   |  pend fav : 552
  known ints : 106/125k, 456/712k, 793/1.27M      | own finds : 6377
  dictionary : 0/0, 0/0, 45/125k                  |  imported : n/a
       havoc : 519/69.8k, 0/0                     | stability : 46.31%
        trim : 0.00%/14.0k, 0.00%                 |
                                                         [cpu000: 40%]
```



A problem repeatedly occurred with "10.108.144.147:8000/id%3A000009%2Csig%3A11%2Csrc%3A000239%2Cop%3Aarith8%2Cpos%3A72%2Cval%3A-12.html"

WASM & Web Browsers

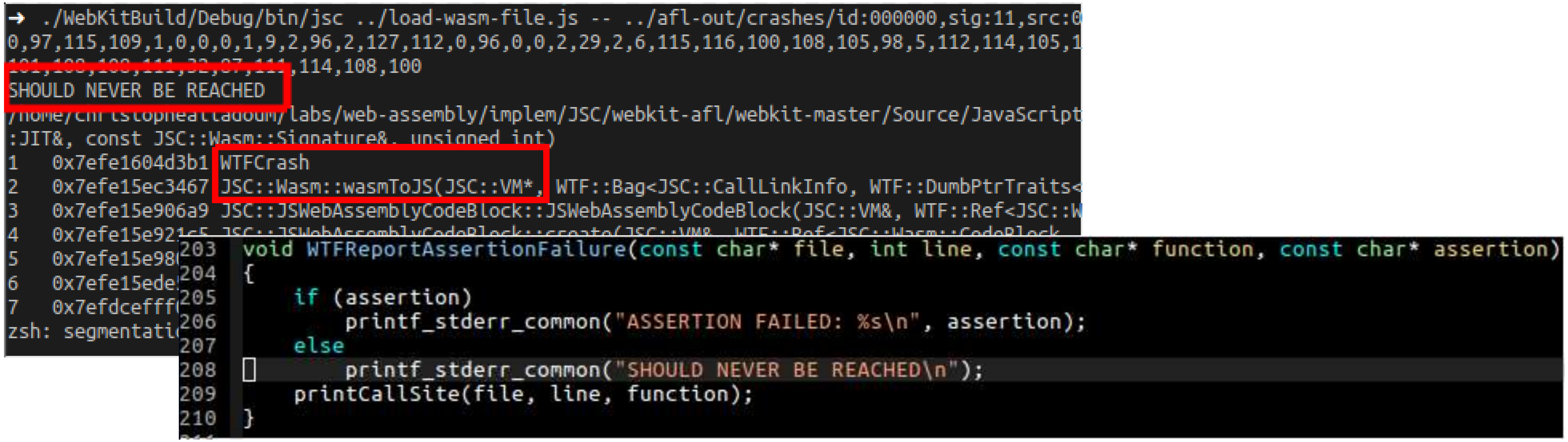
- About ~30 unique crashes



```
➜  ./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 --
```

WASM & Web Browsers

- About ~30 unique crashes
    - But not exploitable … ☹

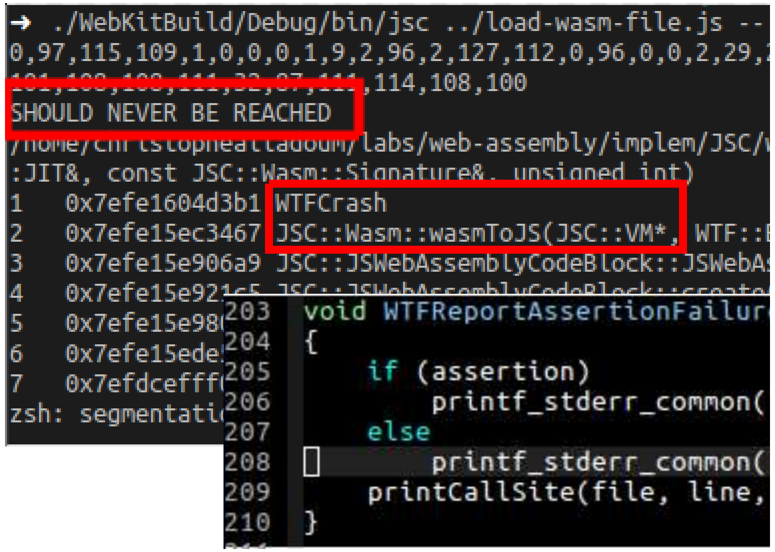


```
➜  ./WebKitBuild/Debug/bin/jsc ../load-wasm-file.js -- ../afl-out/crashes/id:000000,sig:11,src:0
0,97,115,109,1,0,0,0,1,9,2,96,2,127,112,0,96,0,0,2,29,2,6,115,116,100,108,105,98,5,112,114,105,1
101,108,108,111,32,87,111,114,108,100
SHOULD NEVER BE REACHED
/home/christopheattadoum/labs/web-assembly/implem/JSC/webkit-afl/webkit-master/Source/JavaScript
: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   0x7efe15e98
6   0x7efe15ede
7   0x7efdcefff
zsh: segmentati
```

```
void WTFReportAssertionFailure(const char* file, int line, const char* function, const char* assertion)
{
    if (assertion)
        printf_stderr_common("ASSERTION FAILED: %s\n", assertion);
    else
        printf_stderr_common("SHOULD NEVER BE REACHED\n");
    printCallSite(file, line, function);
}
```

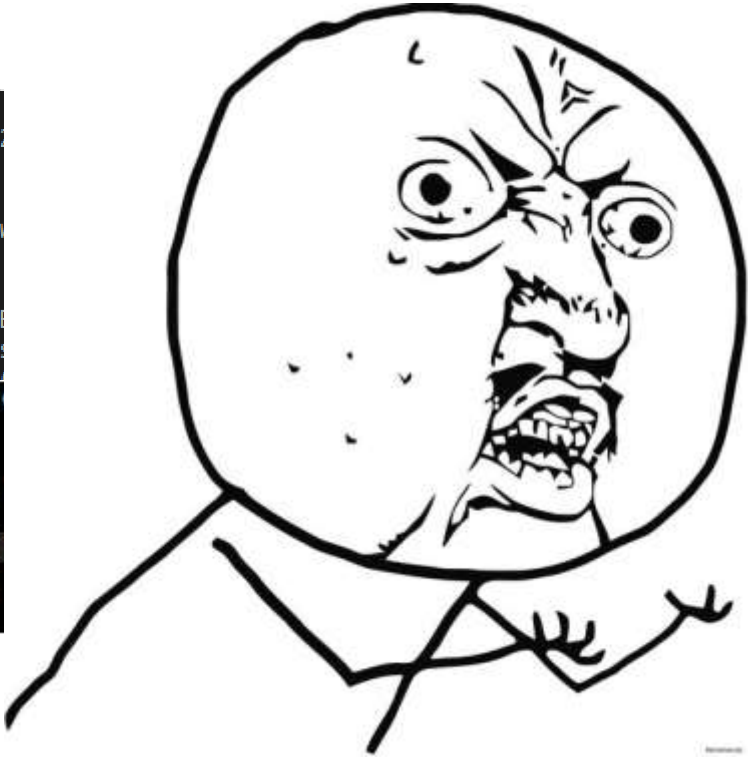
WASM & Web Browsers

- About ~30 unique crashes
  - But not exploitable … ☹



```
➜  ./WebKitBuild/Debug/bin/jsc ../load-wasm-file.js --
0,97,115,109,1,0,0,0,1,9,2,96,2,127,112,0,96,0,0,2,29,2
101,108,108,111,32,87,111,114,108,100
SHOULD NEVER BE REACHED
/home/christopheattadoum/labs/web-assembly/implem/JSC/w
:JIT&, const JSC::Wasm::Signature&, unsigned int)
1   0x7efe1604d3b1 WTFCrash
2   0x7efe15ec3467 JSC::Wasm::wasmToJS(JSC::VM*, WTF::B
3   0x7efe15e906a9 JSC::JSWebAssemblyCodeBlock::JSWebAs
4   0x7efe15e921c5 JSC::JSWebAssemblyCodeBlock::create
5   0x7efe15e98
6   0x7efe15ede
7   0x7efdcefff
zsh: segmentati
```

```
void WTFReportAssertionFailur
{
    if (assertion)
        printf_stderr_common(
    else
        printf_stderr_common(
    printCallSite(file, line,
}
```



```
ion, const char* assertion)
```

WASM & Web Browsers

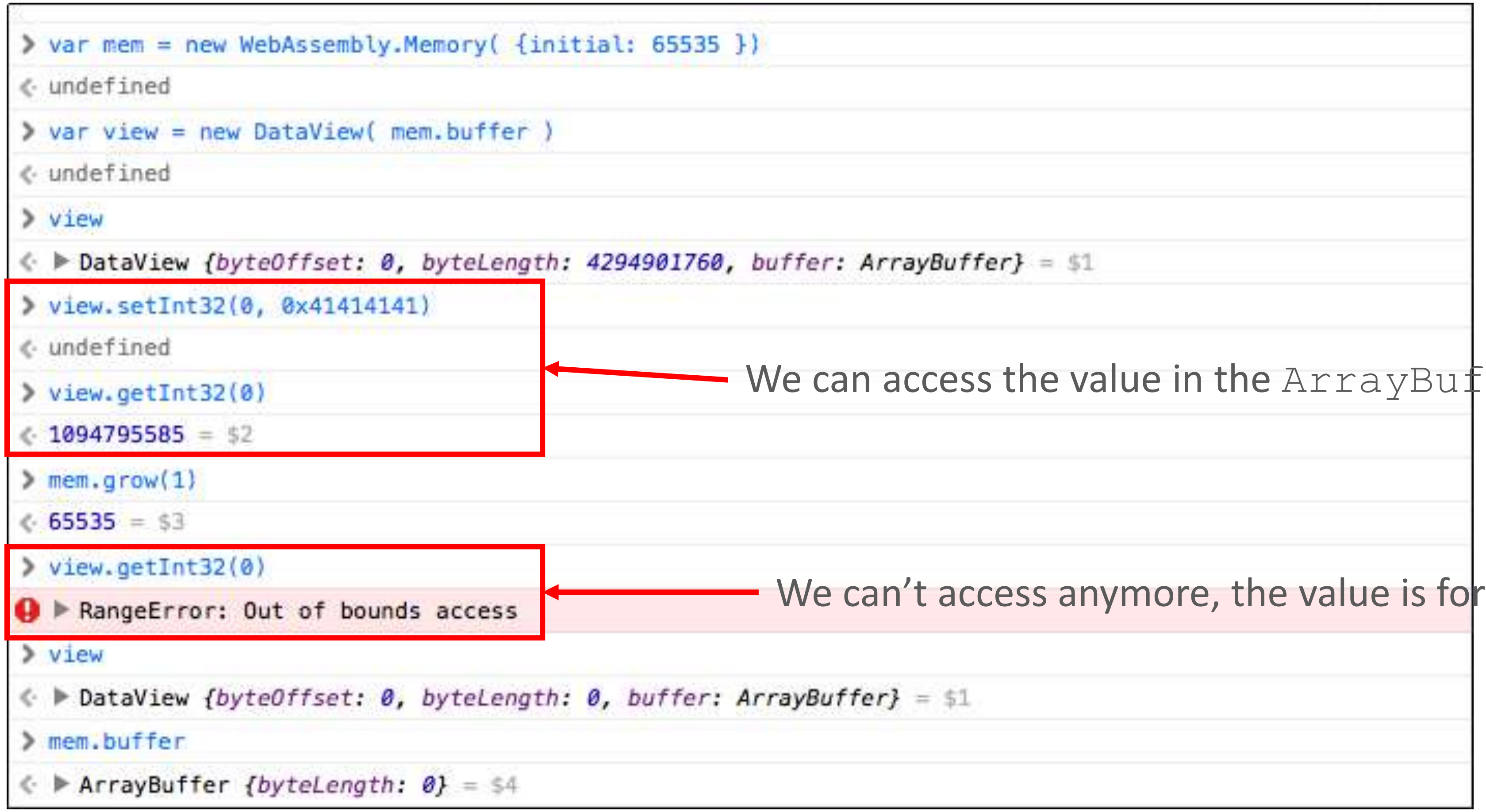
- In fact, most web browsers already provide their own tests/fuzzing scripts
  - All because WASM has a strict binary format
    - Perfect target for AFL/LibFuzzer fuzzing (unlike JS)
    - Can be tested independently from JavaScript

- Security from Simplicity
  - There is some other edge cases to test
  - MVP 1.0 will soon be obsoleted by new release
    - New features (new bugs?)

- Any other attack we could find?
  - Arithmetic errors
  - Denial-of-Service
  - Race conditions in WASM memory byte array (shared with JS)

WASM & Web Browsers

- Example: Arithmetic error in WASM engine in Safari



```
> var mem = new WebAssembly.Memory( {initial: 65535 })
< undefined
> var view = new DataView( mem.buffer )
< undefined
> view
< ▶ DataView {byteOffset: 0, byteLength: 4294901760, buffer: ArrayBuffer} = $1
> view.setInt32(0, 0x41414141)
< undefined
> view.getInt32(0)
< 1094795585 = $2
> mem.grow(1)
< 65535 = $3
> view.getInt32(0)
! ▶ RangeError: Out of bounds access
> view
< ▶ DataView {byteOffset: 0, byteLength: 0, buffer: ArrayBuffer} = $1
> mem.buffer
< ▶ ArrayBuffer {byteLength: 0} = $4
```

We can access the value in the `ArrayBuffer`

We can't access anymore, the value is forgotten

Abusing WebAssembly

- WebAssembly is a good candidate for Side-Channel attacks
  - Although made harder by post-Spectre mitigations
- JavaScript code obfuscation (WAF/AV bypass etc.)

```
eval( MyWasmInstance.exports.SomethingObfuscated() );
```

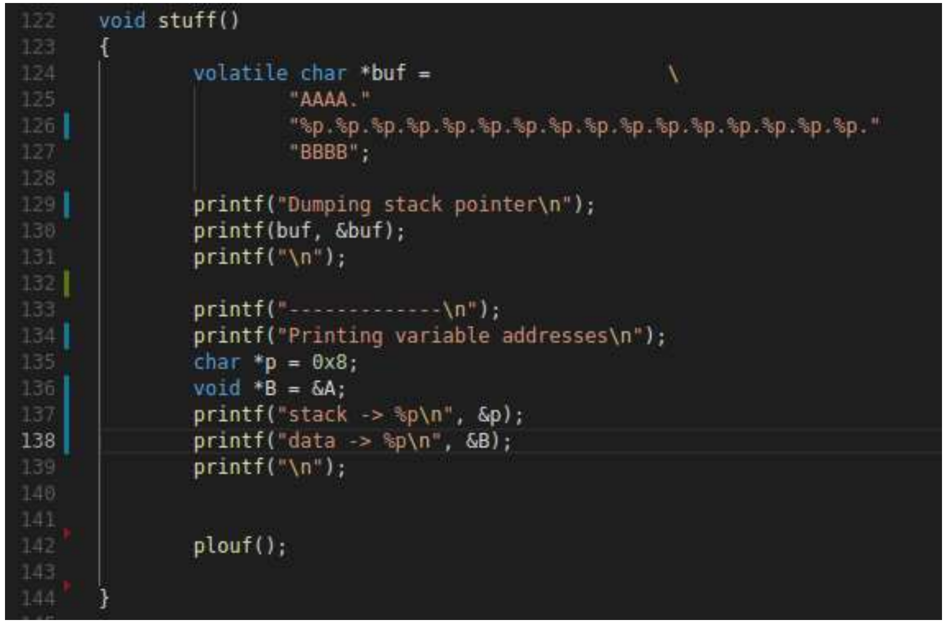
- Excellent use case for (not-so) bad guys: crypto-mining

Crypto Mining is the process of bitcoin mining utilizing remote server(s) with shared processing power.

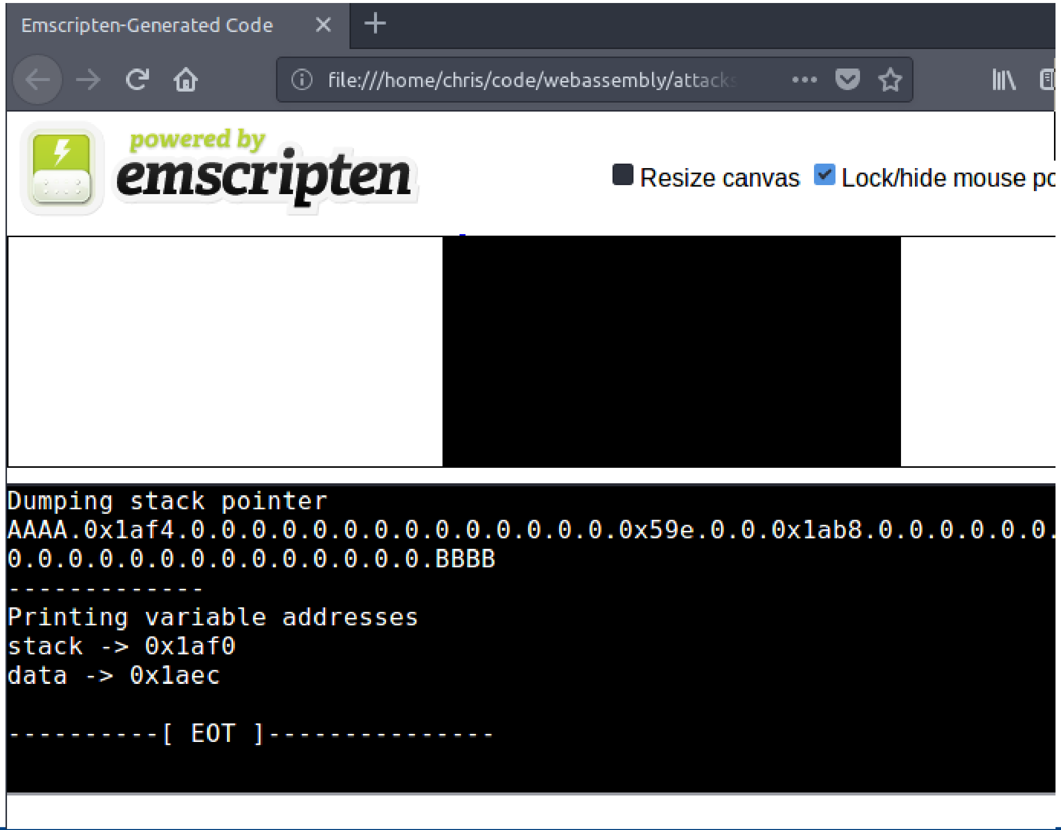
  - Used to be achieved by pure JavaScript code (like CoinHive)
  - Computation resource -> close to native performance with WASM
  - Already a few (Open-Source) variants spotted
    - CryptoNight
    - Xmonash

Abusing WebAssembly

- What about code inside the VM?
  - Still affected by “traditional” C-style attacks
    - Format strings
    - Buffer overflow (stack, “heap”)



```c
void stuff()
{
        volatile char *buf =                        \
                "AAAA."
                "%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p."
                "BBBB";

        printf("Dumping stack pointer\n");
        printf(buf, &buf);
        printf("\n");

        printf("-------------\n");
        printf("Printing variable addresses\n");
        char *p = 0x8;
        void *B = &A;
        printf("stack -> %p\n", &p);
        printf("data -> %p\n", &B);
        printf("\n");


        plouf();

}
```



Emscripten-Generated Code
file:///home/chris/code/webassembly/attack
powered by emscripten
Resize canvas
Lock/hide mouse pc

```
Dumping stack pointer
AAAA.0x1af4.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0x59e.0.0.0x1ab8.0.0.0.0.0.0.
0.0.0.0.0.0.0.0.0.0.0.0.0.0.BBBB
-------------
Printing variable addresses
stack -> 0x1af0
data -> 0x1aec

----------[ EOT ]--------------
```

What about the defense ?

- Disable WASM ?
  - Currently, only with a few tricks on Chrome & Firefox
    - Not possible (yet) on Edge or Safari
    - Firefox
      - Set `javascript.options.wasm` to False in `about:config`
    - Chrome

```
$ chrome -js-flags=noexpose-wasm
```

- Traffic inspection
  - WebAssembly files have a distinct magic number (`‘\x00asm’`)
  - But hard to audit

What about the defense ?

- Static analysis
  - IDA script to disassemble .wasm files
  - `wasm2c` will generate a pseudo-C code (HexRays-decompiler style) from a WASM file

- Complex projects (good or bad) will use a compiler (such as EmScripten)
  - Can be used to flag binaries
    - More complex when WASM embedded in JS
      - But code size limited to 4096
  - Generate AV signatures
    - Risk of FP
    - Hard to analyze dynamically

Future of WASM

(or why we should keep an eye open)

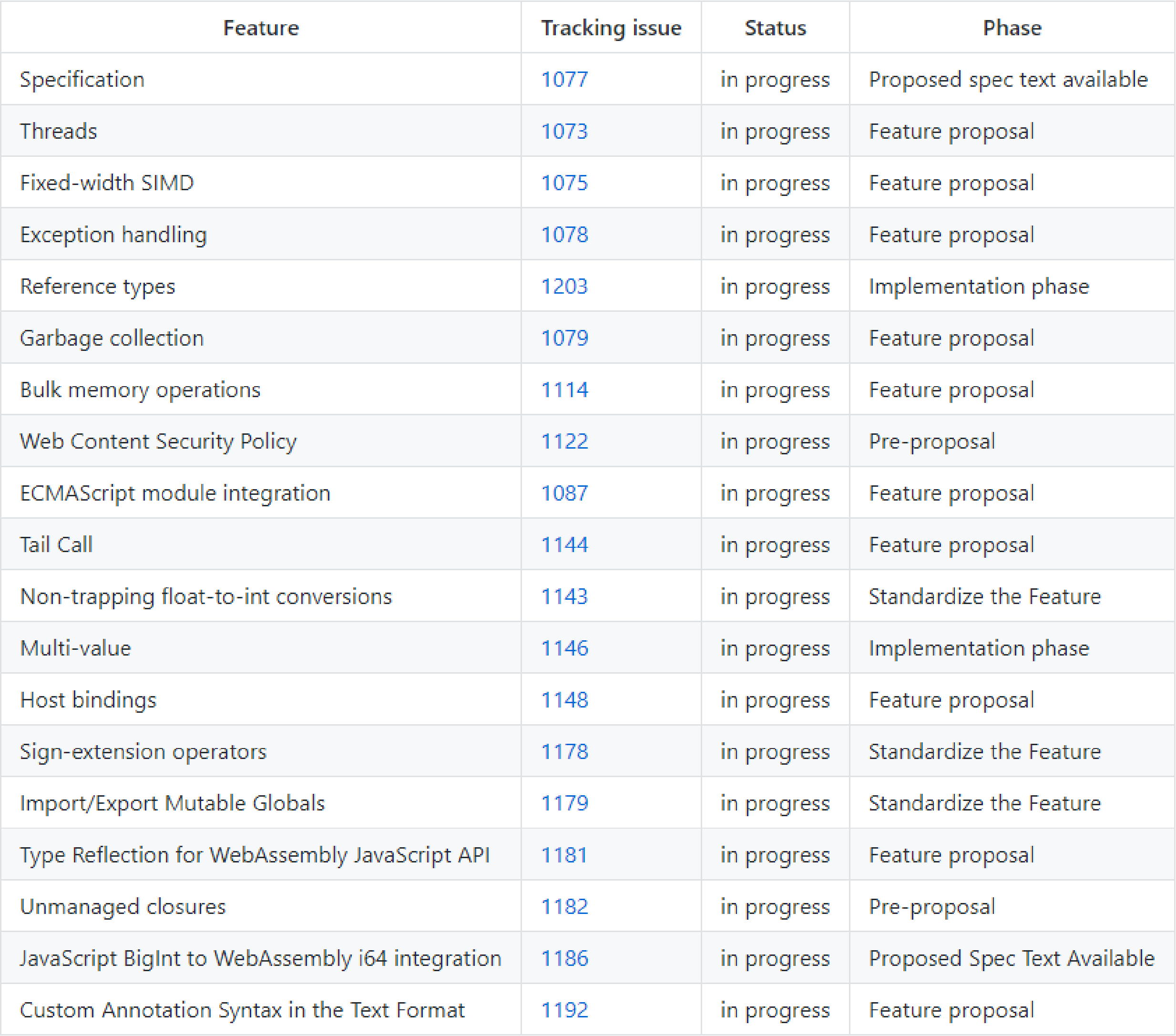
SOPHOS

What’s next?

- Those features explained were specified in MVP 1.0

- WASM standard is still under very active development
  - Overcome some limitations of the existing platform
  - Provide additional features
  - Improve integration with JS / DOM

What’s next?



Feature	Tracking issue	Status	Phase
Specification	1077	in progress	Proposed spec text available
Threads	1073	in progress	Feature proposal
Fixed-width SIMD	1075	in progress	Feature proposal
Exception handling	1078	in progress	Feature proposal
Reference types	1203	in progress	Implementation phase
Garbage collection	1079	in progress	Feature proposal
Bulk memory operations	1114	in progress	Feature proposal
Web Content Security Policy	1122	in progress	Pre-proposal
ECMAScript module integration	1087	in progress	Feature proposal
Tail Call	1144	in progress	Feature proposal
Non-trapping float-to-int conversions	1143	in progress	Standardize the Feature
Multi-value	1146	in progress	Implementation phase
Host bindings	1148	in progress	Feature proposal
Sign-extension operators	1178	in progress	Standardize the Feature
Import/Export Mutable Globals	1179	in progress	Standardize the Feature
Type Reflection for WebAssembly JavaScript API	1181	in progress	Feature proposal
Unmanaged closures	1182	in progress	Pre-proposal
JavaScript BigInt to WebAssembly i64 integration	1186	in progress	Proposed Spec Text Available
Custom Annotation Syntax in the Text Format	1192	in progress	Feature proposal

What’s next?

- MVP allows many extensions including:
  - On-demand memory allocation
    - `mmap()`-like operation (`munmap()` too)
    - Shared memory between multiple running modules
    - Define permission mechanism (`mprotect()`-like)
    - High memory pressure scenario ?
    - Memory overlap ?
    - Will this introduce pointers into WASM ?
  - 64-bit integer support
    - Integer overflow
  - SIMD
    - 128-bit floats
  - Multi-threading
    - Race conditions ?
- Some of those features are already being implemented in Firefox / Chrome

Conclusion

SOPHOS

Conclusion

- WebAssembly is the new kid on the block, we should deal with it
    - More applications will turn to it
        - Active development
        - More frameworks allow for a smooth transition from ASM.js and regular JS
    - Security was not left out
        - Robust specification, simple by design
            - Limits the attack window
        - But doesn’t prevent implementation bugs
    - Keep an eye out for future specifications (and their implementations!)

Useful links

- Presentation + demo source code + custom toolkit
  - https://github.com/Sophos/WebAssembly/

- Specification
  - https://github.com/WebAssembly/design/blob/master/

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

- Bugs exploiting implementations
  - https://android-developers.googleblog.com/2018/01/android-security-ecosystem-investments.html
  - https://bugs.chromium.org/p/chromium/issues/detail?id=766253
  - https://bugs.chromium.org/p/project-zero/issues/detail?id=1522
  - https://bugs.chromium.org/p/project-zero/issues/detail?id=1545
  - https://bugs.chromium.org/p/project-zero/issues/detail?id=1546

Thanks for listening !

--- EOT

SOPHOS

SOPHOS

Cybersecurity made simple.